# Formalizing Relational Databases as OWL Ontologies

By

**Natalia Villanueva-Rosales**

A thesis submitted to
the Faculty of Graduate Studies and Postdoctoral Affairs
in partial fulfilment of
the requirements for the degree of

**Doctor of Philosophy**
in
Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario, Canada

*Your file Votre référence*
*ISBN: 978-0-494-87761-6*

*Our file Notre référence*
*ISBN: 978-0-494-87761-6*

*A mi ejemplo a seguir, mi madre*

*(to my mother, my role model):*

*Profr. Enriqueta Rosales Hernández*

# Abstract

While the World Wide Web provides a platform for sharing information, using it to *answer sophisticated questions* involving significant background knowledge is not currently possible. The principal reason behind this limitation is that digital information, while machine processable, is not machine understandable. In contrast, the emerging *Semantic Web* leverages current web architecture and additionally provides the means by which explicit *semantics* can be assigned to raw data through *ontologies* that explicitly describe and relate objects using *formal, machine understandable* representations. Widespread adoption of the Semantic Web depends significantly on an increase in the number of ontologies spanning different domains of knowledge. Manually designing ontologies, however, is tedious, acting as a *knowledge acquisition bottleneck.*

In this thesis, we present DBOwlizer, a novel framework for the *automatic* design of expressive ontologies from (normalized) relational databases. This approach extracts and accurately represents the information found in a database by examining its composition.

DBOwlizer allows one to: i) represent mappings between a relational database and an OWL ontology, ii) automatically generate such mappings using rule-based heuristics and iii) execute these mappings to generate a populated OWL ontology.

DBOwlizer goes beyond other existing approaches by: i) representing mappings using standardized Semantic Web languages, thereby avoiding the use of application specific non-standard mapping languages, ii) facilitating the modification and reuse of mapping heuristics, iii) enabling the comparison of heterogeneous heuristics that map relational databases to ontologies, iv) automatically generating decidable database-to-ontology mappings for more expressive ontologies, v) separating domain knowledge from model knowledge in the output ontology, and vi) mapping semantics from database views, including a pattern to characterize aggregation functions.

We anticipate that this work will help to expose the rich content currently stored in thousands of databases in a way that users, in particular scientists, can be more effective in retrieving information that supports their scientific endeavours.

**Keywords**

Semantic web, knowledge representation, ontology design, ontology engineering, expressive ontologies, data integration, reverse engineering, question answering.

# Acknowledgements

Many people have been part of my doctoral studies as teachers, colleagues and friends. This work would have not been possible without the steadfast encouragement and unconditional support of my mother Enriqueta Rosales Hernández, aunt Ofelia Rosales Hernández and grandmother Micaela Hernández Macias. I would also like to thank my husband Omar Ontiveros for his patience and support, my baby son Miguel for bringing happiness and inspiration to my life and my sister Mariana Villanueva Rosales, who traveled to Canada to take care of Miguel while I was writing this manuscript.

I owe my deepest gratitude to my supervisor, Michel Dumontier for his encouragement and support during my doctoral studies. Not only did he introduce me to the fascinating area of Life Sciences, but he also spent time explaining the challenges and opportunities in this field. His passion and vision inspired me to work in this area.

I would also like to thank my thesis examiners: Helen Chen, Jeff Dawson, Fazel Famili and Anil Somayaji for their comments which improved the quality of this manuscript. My deepest gratitude also goes to Jean-Pierre Corriveau for his wise advice through all these years at Carleton University.

I was surrounded by knowledgeable and nice people at Dumontier Lab, who helped me on a daily basis, particularly my dear friends Alison Callahan and Jose Miguel Cruz Toledo who were always available for discussions and proof readings of early versions of

this manuscript. Thank you both for your enormous support and encouragement which made the last stage of my life as graduate a pleasant one.

I have been lucky to be surrounded by good friends in this long journey including Kamilla Jóhansdóttir , Gerardo Reynaga, my "compadres" Juan Pablo Zamora Zapata and Martha Camacho de Zamora and Mauricio Vinés, who have been at my side to overcome challenges in my academic and personal life.

Thanks to my friends in Mexico, who have always been with me and kept me as part of their life despite the distance. And also to those who were near, specially my WISE (Women in Science and Engineering) friends for being my *accomplices* in creating CU-WISE.

Last, but not least, I am grateful for the funding received to carry out this research from the Mexican National Council of Science and Technology (CONACYT) scholarship 150581 for doctoral studies, the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant and Canada's Advanced Research and Innovation (CANARIE) Network Enabled Platform 2.

# Table of Contents

# List of Tables

# List of Figures

# Acronyms

| | |
|---|---|
| **1NF** | First Normal Form |
| **2NF** | Second Normal Form |
| **3NF** | Third Normal Form |
| **CWA** | Closed World Assumption |
| **DL** | Description Logics |
| **ER** | Entity-Relationship model |
| **GO** | Gene Ontology |
| **OWA** | Open World Assumption |
| **OWL** | Web Ontology Language |
| **RDB** | Relational Databases |
| **RDBMS** | Relational Database Management System |
| **RDF** | Resource Description Framework |
| **UNA** | Unique Name Assumption |
| **SGD** | Saccharomyces Genome Database |
| **SIO** | Semanticscience Integrated Ontology |
| **SWRL** | Semantic Web Rule Language |
| **URI** | Uniform Resource Identifier |
| **URN** | Uniform Resource Name |
| **WWW** | World Wide Web |
| **W3C** | World Wide Web Consortium |

# Chapter 1 Introduction

Ontologies are the backbone of Semantic Web. Therefore, the promise of Semantic Web to be able to move beyond syntactic information retrieval and enter the realm of question answering (*i.e.* answer to queries enriched and guided by domain knowledge) depends on an increase in the number of ontologies spanning such domains. Towards this goal, we hereby propose an automatic approach for the generation of ontologies from relational databases (RDB) and discuss the advantages of our approach with respect to previous work. In this Chapter, we further describe the motivation, specific objectives and contributions of this work.

## 1.1 Motivation

The World Wide Web provides a platform for sharing and navigating enormous amount of data embedded in web documents. However, computers cannot interpret words, sentences or tables so as to correctly *reason* about the entities and the relations between them that are implicitly stated in those documents (Robu, Robu et al. 2006).

To answer a query on the current web, also known as the *syntactic web* (Maedche and Staab 2001), search engines identify highly connected resources that *syntactically match* a subset of the words contained in the questions. Users then *retrieve* these documents

and search through them until they find the information of interest to them. For instance, consider a scientist interested in yeast, a fungal species whose ability to ferment carbohydrates is fundamental in the production of bread, beer and wine (Raven, Johnson et al. 2002). She may be interested in new avenues for experimental studies about the yeast gene NSA3. Her first attempt to retrieve the latest research discoveries about this gene would be searching using the term "NSA3". Google™ returns about 260,000 links to web pages, publications (in PDF), images, *etc.*, where the keyword "NSA3" is present. In order to retrieve the relevant data, our yeast scientist would have to *manually inspect* each one of the retrieved resources and *manually integrate* it with information from other web pages. While some of these web resources might contain the information needed, an overwhelming majority have information that does not contribute to her goal, for example, all the pages with information about the NSA3 musical band. Refining the search would require the use of additional keywords drawn from *domain knowledge*, *i.e.* searching with "NSA3 gene yeast", returns about 1,790 links. Importantly, our scientist would also have to be aware of other names related to NSA3, such as its standard name CIC1 or the systematic name YHR052W, given the proliferation of identifiers for the same entity in the Life Sciences. A knowledgeable user might also search using a yeast specific search engine like the one provided for the *Saccharomyces* Genome Database (SGD) (Cherry, Adler et al. 1998). This engine returns a large amount of information about the NSA3 gene, organized in different tabs that facilitate navigation, but requires our scientist to know in advance the *type* of information needed. SGD, like other database applications, only allows the

users to *search* for data using predefined queries, which restricts the exploration of data stored in these databases. Moreover, relevant data must be retrieved from distributed sources and *manually integrated.*

Efficient information retrieval is not only important for Life Sciences, but also has important consequences in the delivery of Health Care. New knowledge is emerging with respect to the relationship between adverse side effects, drug efficiency and genetic background. This knowledge promises a new era in medicine called *personalized medicine* in which practitioners tailor a drug and its dosage to individuals to achieve the most effective drug treatment (Evans and Relling 2004). For example, one may ask *is it possible for a doctor to determine the most suitable drug treatment for depression to avoid a specific side effect using my genetic information* (with a simple query)? Until computers are able to efficiently *retrieve* and *integrate* distributed knowledge available (on the web), answering this question will remain a challenge.

The primary goal of the *Semantic Web* is to add *semantics* to the current Web, by designing *ontologies* which explicitly describe and relate objects using *formal* languages that a *machine can understand and process* (Berners-Lee and Hendler 2001; Berners-Lee, Hall et al. 2006). One of these languages is the Web Ontology Language (OWL) (Hitzler, Krötzsch et al. 2009), which allows the creation of ontologies based on Description Logics (DL) (Horrocks 2005). Description Logics is a subset of First Order Logic that allows description of complex concepts from simpler ones with an emphasis on decidability of reasoning tasks (*i.e.* the results will be returned in a finite amount of time). These reasoning tasks are key to question answering along with tapping into

expert knowledge through domain ontologies. For instance, a yeast ontology may not only allow question answering with specific knowledge about yeast, but also facilitates the representation and validation of hypotheses, providing the facts that contribute to its evaluation (Callahan, Dumontier et al. 2011).

Ontologies are generally designed manually and involve the compilation of knowledge from domain experts or heterogeneous sources such as publications, data, images, *etc.* This design can be guided by semantic web best practices (W3C 2004b), normalization (Rector 2003), design patterns and workflows (Aranguren 2005). However, this task is tedious and acts as a *knowledge acquisition bottleneck* (Maedche and Staab 2001).

A step towards the proliferation of ontologies is the automatic generation of ontologies which can take advantage of the knowledge represented in relational databases. A large amount of the data published on the web is stored in relational databases. In fact, the databases used to dynamically generate web documents are estimated to be 400 to 550 times larger than the data available on the web and are collectively referred to as the Deep Web (Bin, Mitesh et al. 2007).

Relational databases are widely used to efficiently store and query data. In relational databases everything is represented by means of a relation, which limits the ability to explicitly represent the differences between entities, their attributes and their relations. However, the structure of a well designed and normalized database contains certain patterns that can be used to recover the initial concepts of the domain being modelled in such a database (*i.e.* domain knowledge).

The idea of analyzing an existing system and recovering corresponding design specifications is not new. In fact, this is the problem tackled by the area of software *reverse engineering* (Winans and Davis 1990). In particular, early works in this area with the same premise propose methodologies for obtaining the entities and relations represented in relational databases (Navathe and Awong 1987; Chiang, Barron et al. 1994) although their goal is not the generation of ontologies.

More recent methodologies and tools have focused on the migration of data to the Semantic Web. The approaches presented in (Shen, Huang et al. 2006), (Astrova, Korda et al. 2007), (Man, Xiao-Yong et al. 2005) and (Stojanovic, Stojanovic et al. 2002) make use of simple heuristics to map databases to ontologies, but these heuristics do not completely exploit the patterns that can be found in a database such as the semantics of queries defined in views. As a result the ontologies generated are not very expressive and do not contain all the information that the database represents. Similarly, ontologies obtained when using tools and frameworks to automatically generate ontologies from databases are not very expressive. While some tools contain their heuristics in the application code such as DataMaster (Nyulas, Oconnor et al. 2007) and ROSEX (Curino, Orsi et al. 2009), others tools such as DB2OWL (Cullot, Ghawi et al. 2007) and D2RServer (Bizer 2003) generate a mapping file that can be manually modified and requires the use of an application dependent markup language. The latter applications are an extension of tools originally created to manually define mappings between databases and existing ontologies.

While these efforts are a first step towards the automated extraction of ontologies we believe that more expressive ontologies can be obtained by undertaking a deeper analysis of the composition of a relational database and taking advantage of the expressivity of OWL.

In this thesis we propose a framework for the representation, automated generation and execution of mappings to create expressive ontologies that preserve the meaning of the database composition patterns as well as the information stored in a database. The resulting ontologies can be considered domain ontologies, which represent the perspective of the database designers and facilitate question answering over the domain. At the very least, these ontologies can serve as building blocks for more complex ontologies.

The migration of data between relational databases and ontologies has gained momentum with the recent creation of the RDB2RDF (Ezzat and Hausenblas 2009) whose mission is to provide a standard mapping language between relational databases and the Semantic Web Languages RDF (Manola and Miller 2004) and OWL. The contributions of this thesis are complementary to this effort by providing insights towards the automatic generation of database to ontology mappings. These contributions, along with the hypothesis and objectives are described in the following sections.

Our work sets the stage for vastly improving the representation of the semantics of data stored in relational databases, which will become a cornerstone for large scale data integration of Semantic Web data.

## 1.2 Hypothesis

Our hypothesis is that an expressive OWL ontology can be automatically created using a set of rule-based heuristics represented in Semantic Web Languages over a normalized relational database.

## 1.3 Objectives

The primary objectives of the proposed research are:

1. Represent mappings between relational databases and OWL ontologies using Semantic Web Languages: OWL and SWRL.
2. Automatically generate mappings from relational databases to expressive OWL ontologies.
3. Improve previous approaches by extracting the information represented in database views.
4. Enable the automatic population of ontologies from relational databases.

The secondary objectives of the proposed research are:

- Provide a framework for comparison of methodologies aiming to generate OWL ontologies from relational databases.
- Develop an application to facilitate the automatic generation of OWL ontologies from relational databases.

## 1.4 Contributions

The major contributions of this thesis are:

1. The design of an ontology to capture the composition and semantics of relational databases.
2. The design of an ontology to capture mappings between elements of relational databases and ontologies.
3. The use of Semantic Web languages for the representation of mappings between relational databases and ontologies.
4. Rule-based heuristics for the automatic generation of an OWL ontology from a relational database.
5. Representation of relational database queries in OWL ontologies, including aggregation functions.
6. DBOwlizer, a framework for: i) the execution of mappings to create OWL ontologies from relational databases to OWL ontologies and ii) the comparison of approaches that generate ontologies from relational models.
7. A comparison of DBOwlizer against related approaches with the use of an implementation prototype.

This thesis also contributes to the adoption of Semantic Web technologies by automating the process of generating and populating ontologies from relational databases.

The generation of mappings between databases and existing ontologies is out of the scope of this thesis.

This document is organized as follows: Chapter 2 provides an overview of relational databases and ontologies along with their differences and similarities. Chapter 3 introduces the process of manually formalizing knowledge representation using ontologies. The challenges of this task are illustrated with the creation of three bio-ontologies as a preamble to Chapter 4, which describes DBOwlizer, a framework for the automatic mapping of ontologies. A prototype to illustrate the features of this framework is presented in Chapter 5. This prototype is used to implement heuristics of related tools and frameworks that have the same goal as DBOwlizer. The results of comparing related work with DBOwlizer are presented in Chapter 6, followed by the conclusions and future work in Chapter 7.

# Chapter 2 Background

In this thesis we address the automatic mapping of information stored in relational databases to Semantic Web ontologies. Therefore, this chapter describes the components and logics behind Semantic Web and relational databases to represent data and knowledge.

## 2.1 Relational Databases

Relational databases are a rich source of information worldwide. Our daily activities may involve interacting with a database. A simple task like buying an item in a grocery store or online will generate an update in the store inventory, and if we use a credit card, it will also involve a transaction to notify our bank that we have subtracted certain amount from our bank account. Relational databases play an important role in almost all the applications where computers are used: entertainment, business, education, science, health care, etc. A database is designed for a specific purpose, represents some aspect of the world (universe of discourse) and is a logically coherent collection of facts with some inherent meaning (Elmasri and Navathe 2003). Therefore, when a change in the world happens, this change should be reflected in the database. This is one of the

tasks of a Relational Database Management System (RDBMS), a piece of software that allows us to create and maintain a database.

Creating a relational database involves the specification of the data that will be stored such as names, data types, relationships and constraints to guarantee the consistency of the data at any time. This information is specified in a database schema, which is described in the following subsection.

### 2.1.1 Database schema

In referring to database schemas, we consider conceptual schemas which, according to Elmasri and Navathe (Elmasri and Navathe 2003), hide the details of physical storage structures and concentrate on describing entities, data types, relationships, user operations and constraints. A database schema diagram is a graphical representation of a database schema, see for example the employees database schema diagram in Figure 9.

A standard language to describe and manipulate (query) relational databases is the Standard Query Language (SQL) (Eisenberg and Melton 1999). Although in practice RDBMSs vendors may modify or extend this language, the basic concepts are the same. In this manuscript we use the SQL syntax variation of MySQL (Widenius, Axmark et al. 2002).

Relational database components are described by means of CREATE statements. The first component created in a database schema is the database itself. For instance,

consider a database that contains information about employees. The following CREATE DATABASE statement creates a database schema called **employees**.

```
CREATE DATABASE `employees`;                                    (1)
```

### 2.1.2 Database tables

One of the key elements in databases are tables. A table groups a set of rows with the same attributes, each of those is organized in columns. Consider for instance Table 1, which contains information about specific employees. It contains the columns id, name, address and salary and a total of 4 rows.

Table 1. A table that groups information about employees

| id | name | address | salary |
|---|---|---|---|
| 1 | Joe | OTTAWA | 100,000 |
| 2 | Peter | OTTAWA | 50,000 |
| 3 | Natalia | OTTAWA | 100,000 |
| 4 | Mary | MEXICO | 25,000 |

**Rows**

**Columns**

In SQL, a table is created through the CREATE TABLE statement. For instance, the SQL statement to create the table **employee** (Table 1) is described as follows:

```
CREATE TABLE  `employees`.`employee` (                          (2)
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(45) NOT NULL,
  `address` varchar(100) NOT NULL,
  `salary` float unsigned NOT NULL DEFAULT '25000',
  PRIMARY KEY (`id`),
  UNIQUE KEY `NameIndex` (`name`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=latin1;
```

Note in (2) that the name of the table is preceded by the name of the database to explicitly relate them. Similarly, a common practice for naming columns is to precede

the name of the column with the table name, with the exception that the name of the database schema is not included in the name of the table. This naming convention will be used in the rest of the document, *e.g.* **employee.id** to refer to the **id** column of the table **employee**.

For readability purposes, we use the bold **Calibri font** to refer to relational database components except when used in figures, tables or formulas in this manuscript.

### 2.1.3 Database columns

The values that a column can hold are constrained by the column's data type and character length. Data types can typically be *numeric*, *string*, *boolean*, *date* and *time* and may vary in different RDBMSs. In our particular example (2), **employee.id** should contain only integer values with a maximum of 10 characters in length. Further restrictions can be imposed on a column, for instance, **the employee.id** column does not allow **NULL** values, while the default value for **employee.salary** column is 25,000 (dollars). Sometimes, an applicable value for a specific column may not be known. For instance, we may not be able to retrieve the address for a specific employee. A special value, called **NULL** is created for these cases. In contrast, when the value of a column is required, the restriction **NOT NULL** is stated. Similarly if a column contains a default value, this value will be used by default until a different value is inserted in the column.

### 2.1.4 Database constraints

Constraints are also represented in the CREATE TABLE statement. These constraints include primary and foreign key. A primary key identifies the entity or relation described

in a row. Primary keys are composed of columns that must hold unique values for each row. For instance, the **employee's** primary key is composed of the column **employee.id** in (2).

Foreign keys restrict the values hold in columns among tables. A foreign key is described with a name, the set of columns that compose it and the set of columns it references. A foreign key usually references the primary key columns of another table. For instance, consider the foreign key described in (3) , which is imposed on the table manager whose complete CREATE TABLE statement is presented in (12) on page 65. The foreign key **FK_manager_proj**, is composed by the column **manager.project** and references the **project.id** column. Informally, this constraint indicates that all the values in the column **manager.project** should also be found in the column **project.id**.

```
KEY `FK_manager_proj` (`project`),                                                    (3)
CONSTRAINT `FK_manager_proj` FOREIGN KEY (`project`) REFERENCES `project` (`id`),
```

By default, MySQL creates indexes for the columns that compose a foreign key and are not part of the primary key. These indexes are described using the word KEY. This naming convention is different from the use of KEY in the relational databases referring to columns with unique values (Elmasri and Navathe 2003). Indexes are specialized data structures represented in files that are used to improve the search for a desired row. They are at the implementation level of the relational database and do not contribute to the conceptual description of it. For example, the statement **KEY `FK_manager_proj` (`project`)** in (3) indicates that an index is created using the column **manager.project**, but it is not implied that this column should have unique values. This index is

automatically created for quick access to the values in **manager.project** given that the column is part of the foreign key **FK_manager_proj**.

Additional table options dealing with implementation characteristics are also included in a table description, such as the charset (character encoding) and the RDBMs used.

Additional constraints cannot be represented in the database schema such as "the salary of an employee cannot be higher than the salary of his manager". These constraints may be represented with triggers or stored procedures, which are pieces of code that are part of the application that uses the relational database. These constraints are represented in non-standard and heterogeneous languages.

### 2.1.5 Database views

Another type of tables are those called virtual tables or views, which are also part of the SQL CREATE statements. A *view* may be a subset of the database or data *derived* from the database which is of particular interest to users. A view is described by means of an SQL query (Elmasri and Navathe 2003). Views are defined by queries that include conditions (imposed to values of specific columns), projections (*i.e.* only a subset of the columns) and aggregation functions (*e.g.* such as the count of rows that satisfy certain conditions). For instance, a user may only be interested in obtaining the names and salaries of employees whose salary is greater than 80,000 (dollars). A view to obtain this information is defined as follows:

```
CREATE VIEW
`employees`.`high_salary_employee_salary` AS
select
`employees`.`employee`.`name` AS `name`,
`employees`.`employee`.`salary` AS `salary`
from `employees`.`employee`
where (`employees`.`employee`.`salary` > 80,000)
```
(4)

For example, the **high_salary_employee_salary** view in the employees database defined in (4) is composed of the columns **high_salary_employee_salary.name**, and **high_salary_employee.salary**. This virtual table is composed by rows of the table **employee** with a value equal or greater to 80,000 in the **salary** column as listed in Table 2.

Table 2. A view (virtual table) that contains the name and salary of employees with a salary greater than 80,00 as defined in (4).

| name | salary |
|---|---|
| Joe | 100,000 |
| Natalia | 100,000 |

The relational (data) model first introduced by Codd (Codd 1970) is the underlying model of relational databases. This model will be covered in the following subsection.

### 2.1.6 Relational model

The principal element in the relational model is the relation. Under the relational model, a table is considered a relation that is composed by a set of tuples (rows) that have identical attributes (columns). Unlike database tables, relations do not allow duplicates and are unordered given their *set* nature (Elmasri and Navathe 2003). Every tuple in the relational model represents a fact about an entity or about a relationship between entities. Attributes have domains, which are analogous to column datatypes and are

restricted by and *not null* constraints and default values (as defined in Subsection 2.1.3). Attributes that represent the same entity do not necessarily have the same name in different relations.

For instance, (5) contains the relation corresponding to the table employee described with the CREATE TABLE statement in (2). This relation contains the attributes employee.id, employee.name, employee.address and employee.salary. Underlined attributes denote that they are part of the primary key.

In the relational model database schema is a collection of relations and integrity constraints (Elmasri and Navathe 2003). Integrity constraints includes entity integrity and referential integrity constraints.

Employees.employee(id,name,address,salary) (5)

In the relational model a candidate key is a set of attributes restricted by the **UNIQUE** constraint, which states that these attributes should not have the same value for distinct tuples. A primary key is a candidate key. The entity integrity constraint additionally impedes primary keys to have NULL values in order to allow the identification of the entity represented by the tuple. Therefore, attributes that are part of the primary key must satisfy the **UNIQUE** and **NOT NULL** constraint.

The interrelation referential integrity constraints are imposed between two relations and are used to maintain consistency among tuples from these two relations. The referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an existing tuple in the referred relation (Elmasri and Navathe 2003). In practice, referential integrity constraints are represented by foreign keys.

Other type of constraints are called semantic integrity constraints such as the one presented in the previous subsection: "the salary of an employee cannot be higher than the salary of his manager". These constraints must be enforced outside the relational model using triggers and stored procedures as previously discussed.

Functional dependencies constraints describe functional relations between two sets of attributes X and Y. This constraint specifies that the value of X determines the value of Y in a relation and is denoted as X -> Y (Elmasri and Navathe 2003). Functional dependencies are used to normalize databases, which is described in the following subsection.

In this document, the relational model terminology is used when we are referring to the relational model itself. For example, we use the words relations, attributes, keys, tuples, etc. for describing the relational-model ontology in section 4.2.

In contrast, we use the SQL nomenclature for relational databases when we refer to specific databases, which instantiate the relational model. For instance, we use tables, columns, rows, *etc.*, to describe the employees database in section. 4.1.

### 2.1.7 Normalization

When designing a relational database, a parameter to determine that the relations in the database are used properly to represent the domain of discourse is their level of normalization (Elmasri and Navathe 2003). Normalization requires, for example, that the attributes of a relation indeed correspond to the entity or relationship being modelled in the relational model. The normal form of a relation, indicates the normal form

condition that it satisfies. Conditions for the First, Second and Third Normal Form (Elmasri and Navathe 2003) are following discussed.

**First Normal Form (1NF)**

A relation is in First Normal Form (1NF) if and only if each attribute of the relation is atomic, which means that composite attributes that contain more than one value are not allowed. The relation Employees.employee in (5) is in First Normal Form.

**Second Normal Form (2NF)**

Second and Third Normal Form are based on the concepts of functional dependency previously described. A relation is in Second Normal Form if it is in First Normal Form and each attribute that is not composing the primary key is fully functionally dependent on the relation's primary key and not a subset. The relation Employees.employee described in (5) is in Second Normal Form. In general, relations without a composite primary key (*i.e.* a primary key composed by two or more attributes) are in Second Normal Form. The Employees.employee_2 relation in (6) is not in Second Normal Form because the attributes employee_2.address and employee_2.salary can be determined by a subset (*i.e.* the attribute employee_2.id) of its primary key composed by the attributes employee_2.id and employee_2.name.

Employees.employee_2(id,name,address,salary) (6)

**Third Normal Form (3NF)**

A relation is in Third Normal Form if it is in Second Normal Form and every non key attribute (*i.e.* an attribute that is not part of a key) is only determined by key attributes.

The relation Employees.employee in (5) is in Third Normal Form, but the relation Employees.employee_3 in (7) is not because the attribute emloyee_3.dependent_age depends on the attribute dependent_name which is not a key attribute.

Employees.employee_3(id,name,address,salary, dependent_name, dependent_age) (7)

If a database designer followed design recommendations, including normalization, the representation of the implicit semantics of the domain of discourse in the relational database are easier to understand by means of finding design patterns.

A more abstract level of representation for the relational model is the entity-relationship model, which is usually one of the first models created in the process of designing a database.

### 2.1.8 Entity-Relationship model

The entity-relationship (ER) model is a high level conceptual model used to describe entities and the relationships that hold amongst them. This model is normally used in the first stages of the database design, after defining the database requirements, to represent the domain on discourse being modelled. The following definitions conform to those given in (Elmasri and Navathe 2003).

In the entity-relationship model, an *Entity* defines a collection (or set) of entities that share the same attributes, it is described by its name and attributes. Entities are uniquely identified by an attribute or set of attributes called *key attributes*. An entity whose key attributes belong to the entity itself, is called *strong entity*. For instance, if the key attribute of an employee is his id and this attribute is only part of the employee entity, employee can be classified as a strong entity. On the contrary, a dependent who

is identified by his name plus the id of the employee on whom he depends may be classified as a *weak entity* whose owner is the employee entity.

In the entity-relationship model, relations between entities are modelled by *relationship*. The *degree* of a relationship is the number of entities participating in such a relationship. Constraints can be applied to limit the possible combinations of entities that participate in a relation; the most common are cardinality ratio and participation. The cardinality ratio for a binary relation constrains the number of relationships that an entity can participate in, *e.g.* (1:N), while participation constraints specify whether the existence of an entity depends on its relationship to another entity. If an entity should participate in a relationship, it has a total participation in the relationship (also known as existence dependency); otherwise it is called partial participation. The possible cardinality ratios for binary relationship types are 1:1, 1:N and M:N. Relationships can also have attributes.

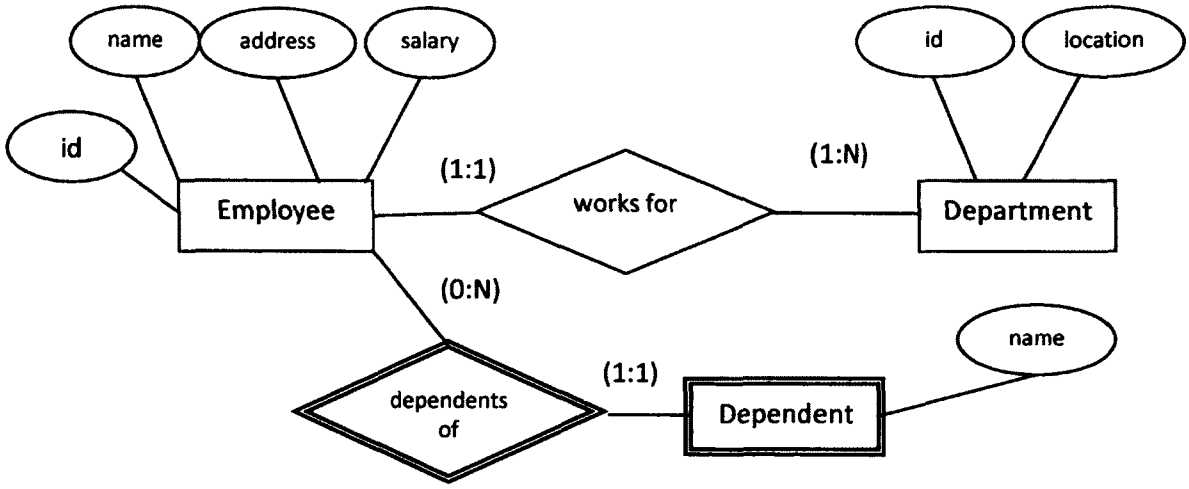



Figure 1. Entity-relationship model diagram for a company database schema. Rectangles represent entities, diamonds represent relationships and ellipses represent attributes. Double lines represent weak entities/relationships. Cardinality constraints are denoted in parenthesis.

Figure 1 is a diagram of the entity-relationship model that represents the strong entities Employee and Department and the weak entity Dependent. The entity Employee contains the attributes id, name, address and salary; the entity Department contains the attributes id and location and the entity Dependent contains the attribute name. The binary relationship "works for" between the Employee and the Department entity is constrained by the cardinality (1:1), which indicates that an employee can work for only one department and the cardinality (1:N), which states that a department may have one or more employees. Similarly, an employee may have zero or more dependents while a dependent only depends on one employee.

In the extended entity-relationship model (EER), class hierarchies with the *is-a* and *part-of* relations are allowed. For instance, a manager *is-a* type of employee.

In the relational model, only *relations* are available to represent both entities and relations. According to the entity-relationship-to-relational mapping algorithm of (Elmasri and Navathe 2003), strong entities are represented by a relation that includes all the simple attributes of an entity. The primary key of the relation represents its key attributes. If the primary key of a relation does not contain a key of another relation, it represents a *strong entity*. A relation can contain also keys from other relations as a proper subset of its key. The keys in this subset are also foreign keys. When this subset contains the key of only one relation, that relation can be recognized as the owner entity and the relation containing the subset represents a *weak entity*. Relationships are represented either by an attribute and a foreign key. Binary relations with M:N

cardinality can be represented as an independent relationship with two attributes, where each attribute is part of a foreign key that references the two entities participating in the relationship. These recommended mappings from the entity-relationship model to the relational model illustrate how entities and relationships are properly represented in the relational model to overcome the limitations of this model. These mapping rules are building blocks for the default heuristics in the DBOwlizer framework.

The following section introduces the Semantic Web.

## 2.2 The Semantic Web

> *"The Semantic Web is about two things. It is about common formats for integration and combination of data drawn from diverse sources, where on the original Web we mainly concentrated on the interchange of documents. It is also about language for recording how the data relates to real world objects. That allows a person, or a machine, to start off in one database, and then move through an unending set of databases which are connected not by wires but by being about the same thing"* (W3C 2001).

The Semantic Web aims to link together the world's knowledge such that we may be able to move beyond *information retrieval* and enter the realm of sophisticated *question answering*. The primary goal of the *Semantic Web* is to add *semantics* (meaning) to the current Web by developing a common set of standards to *formalize* knowledge using *logic-based* representations (Berners-Lee 2000). Effectively, the semantic web markup languages (Figure 2) put forward by the World Wide Web Consortium (W3C) (McCollum, Larsen et al. 2006), provide vocabularies with clear

semantics that a *machine can understand* and *process*. Specialized software programs called *automated reasoners* may then *infer implicit knowledge* from existing information. Users can publish and query machine understandable information using semantic web markup languages. The Semantic Web promises to facilitate the representation, integration and management of data on the web. Thus, *semantically enhanced* applications used by scientists and medical practitioners could automatically discover relevant knowledge in order to answer questions that require domain knowledge. This allows the integration of data from different applications, mapping between different formats, languages and platforms.

### 2.2.1 Semantic Web Languages

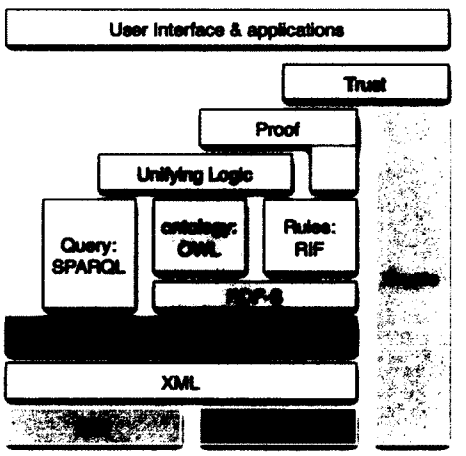



Figure 2. The Semantic Web Language stack from (Berners-Lee 2009).

The Semantic Web framework provides best practices and recommendations over a stack of standardized languages (Figure 2). The Semantic Web is built upon a common character representation (Unicode) (Aliprand, Allen et al. 2004) and naming syntax (Universal Resource Identifier - URI) (Berners-Lee 1994). The *eXtensible Markup*

*Language (XML)* (Bray, Paoli et al. 2000) can be used to create documents containing data delimiting tags, which differs from the HyperText Markup Language (HTML) (Raggett, Le Hors et al. 1999) that focuses only on the presentation of the data. XML documents contain (semi structured) data (in the form of elements) with unambiguous meaning defined using namespaces and use a schema language (XMLS) (Fallside and Walmsley 2004) for data exchange, data integration and document validation.

The *Resource Description Framework (RDF)* (Manola and Miller 2004) allows the generation of data models in terms of resources and relations between them, called *statements, e.g.* "Manager is an Employee". Each statement is composed of a subject, a predicate and an object. In the previous example, "Manager" is the subject, "is a" is the predicate and "Employee" is the object. RDF has been used to publish and link data (Bizer, Heath et al. 2009). The *RDF schema (RDFS)* (Brickley and Guha 2003) defines the notion of a *class* in RDF as a set of resources (that share certain characteristics), also known as instances. RDFS allows the construction of class hierarchies with the *is-a* relation and property hierarchies with the *sub property of* relation. Notice that classes and instances are only allowed to be subjects and objects in a statement, while a property can be a subject, a predicate or an object. Additionally, a blank node in RDF represents a resource for which a URI or literal is not known and is called an anonymous resource.

RDF statements can be represented in XML syntax (RDF/XML), where resources are identified by their URI. RDF/XML is the preferred syntax for applications. An RDF

statement can be also represented as a graph. In a graph representation, subject and objects are nodes and predicates are edges (see Figure 3).

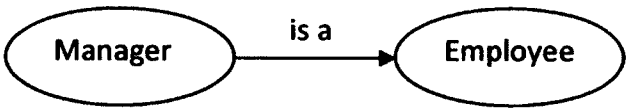



Figure 3. Graph representation of the RDF statement "Manager is a Employee".

Another representation of an RDF statement is a *triple*, (Subject, Predicate, Object) *e.g.* (Manager, is a, Employee), which is the preferred syntax for triple stores. Triple stores, like Sesame (Broekstra, Kampman et al. 2002) and Jena (Brian 2002) provide a framework to store and query RDF data models.

Annotations of web resources using Semantic Web markup languages are the building blocks for ontologies, the key element of the Semantic Web. An *ontology*, as defined by Gruber (Gruber 1993), is a "(shared) specification of a (domain knowledge) conceptualization". The main elements of an ontology are: i) *classes or concepts,* a set of individuals that satisfy the membership requirements of a class, ii) *individuals,* and iii) *relations or properties* that relate individuals.

Although RDFS is recognized as a language for ontologies it is not expressive enough to describe entities with sufficient detail. Some useful constructs to build complex concepts (classes), such as disjoint axioms (*e.g.* an employee is not a project) and cardinality restrictions on properties (*e.g.* an employee can work in at most one department) are missing. A more expressive language is needed.

The next layer built over RDF/RDFS in the Semantic Web stack is the *Web Ontology Language (OWL)*. In this thesis, the term "OWL" refers to the latest W3C recommendation OWL 2 (Hitzler, Krötzsch et al. 2009). OWL is a language that can formally describe the meaning of the terminology used to annotate Web resources. OWL provides additional vocabulary to formally describe entities by generating ontologies such that they become amenable for more advanced automated reasoning as compared to XML, RDF and RDFS. An overview of OWL is provided in the following subsection.

At the same level as OWL in the Semantic Web stack (Figure 2) are the Rule Interchange Format (RIF) (Kifer 2008) and SPARQL (Prud'Hommeaux and Seaborne 2008) . They are both W3C recommendations for the exchange of rules and representation RDF queries respectively. SPARQL can be used to express graph queries across diverse data sources, such as the graph represented in Figure 3. The results of SPARQL queries can be seen as RDF graphs satisfying the restrictions of the query.

The last two layers in the Semantic Web are aimed to support services for agent negotiation based in proof and trust, but these are still under active development by the W3C and no recommendations have yet been put forward at the time of writing this thesis.

A more comprehensive description of Semantic Web ontologies is provided in the following subsection.

### 2.2.2 Web Ontology Language (OWL)

Ontologies formalize the meaning of terms through axiomatic descriptions. In particular, we refer to Semantic Web ontologies as those conceptualizations represented in Semantic Web Languages. These ontologies contain data on the web and are accessible to machines, so that they may be automatically reasoned over.

In particular, an OWL ontology can be seen as a collection of axioms. Axioms can be said to be true or false, which distinguishes them from entities and expressions; classes, properties and individuals are entities which can be combined into expressions defining new entities (Hitzler, Krötzsch et al. 2009). In this document, we denote with bold Bitstream font domain specific terminology, except when used in figures, tables or formulas for readability purposes. The following definitions conform with those in (Hitzler, Krötzsch et al. 2009).

#### OWL Classes

Individuals that share the same attributes are grouped in a *class* and are called *instances* of such a class. For example, the class Employee groups all the employees in a company. Classes can be arranged in *is-a* hierarchies. For example, consider the class Manager (containing the managers in a company), and the assertion Manager is-a Employee. These statements denote that Manager is a *subclass of* Employee. In other words, all the managers are also employees. A diagram with a set representation can be found in Figure 4. OWL contains two predefined classes: *Thing,* which defines is the group of all

the individuals, and *Nothing* which defines the empty set (W3C 2004a). Classes are identified by a unique URL.

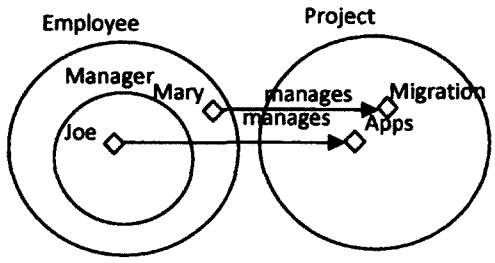



Figure 4. A set representation of classes (circles), individuals (diamonds) and properties or relationships between individuals (arrows).

A class can be named or could remain anonymous (*i.e.* for which only the logical description is provided). As a central feature of OWL, classes can be combined into *expressions* using *constructors* (Hitzler, Krötzsch et al. 2009). As a basic example, the atomic classes Employee and Client can be combined as a union to describe the class of Person. The latter would be described by an OWL class expression, which could be used in statements or in other expressions. In this sense, expressions can be seen as new entities defined by their structure. These expressions are also called *complex classes*.

**OWL properties**

OWL *properties* can relate an individual to another individual, denoted as object properties or an individual with a specific type of data, denoted as data properties. For instance, in Figure 4, manages is an object property that relates instances of the class Manager with instances of the class Project. The domain and range of a property can be asserted. For instance, in Figure 4, the domain of the property manages is Employee and its range is Project.

In OWL, an object property can be (in the logical sense): transitive, reflexive, irreflexive, asymmetric, functional, inverse functional, inverse, equivalent to another property, symmetric or inverse symmetric.

A property can be organized into property hierarchies (sub-property of) and property chains. For instance, if we define the object property supervises as a *sub property of* manages, all the individuals related with the property supervises can be inferred to be related with the property manages. Similarly, if we create the property chain manages o 'has part' -> manages, where o denotes a property chain and it is asserted that Joe manages Apps and Apps 'has part' Requirements, then it can be inferred that Joe manages Requirements.

A *data property* relates an individual with a data value. Data values are restricted by XML Schema data types, such as xsd:float, xsd:string, *etc.* Data values can also have a customized range that can include an enumeration of values. Data properties can be asserted as functional or disjoint (in the logical sense), and can be arranged in a data property hierarchy.

Finally, a*nnotation properties* are used to encode information about the ontology itself (like the author and human-readable label of a class).

**Restrictions**

Necessary conditions for class membership denote the conditions that all individuals that are members of that class should satisfy. For instance we can state as a necessary condition that each member of the class Manager manages some Project. In other

words, all the individuals classified as Manager, should be linked to an individual of project through the property manages.

The membership to a class, in other words, the conditions that an individual has to satisfy in order to be classified as an instance of a class, may be defined by a set of logical descriptions that combine classes and properties using logical connectors. These connectors include *union, intersection, negation and existential/universal quantifiers.* For example, the OWL class Manager can be defined as Employee that manages some Project. Therefore, all the instances of the Employee class that are linked to an instance of Project with the property manages can be inferred as instances of the class Manager. In Figure 4, the individual Mary can be *inferred* as an instance of the class Manager.

Properties can also be restricted by cardinality, existential and universal restrictions. For example the restriction, Employee that manages some Project impose an existential restriction on the class Employee. Individuals satisfying this restrictions are instances of the class Employee and are related to at least one instance of Project with the manages property. In contrast, individuals satisfying the restriction Employee that manages only Project are related only to individuals of Project through the manages property. Also, individuals satisfying the restriction Employee that manages min 2 Project are instance of the class Employee and are related to at least 2 instances of the class Project through the manages property.

**OWL Full and OWL Profiles**

OWL can be considered as a syntactically restricted version of OWL Full (Motik, Patel-Schneider et al. 2009) in order to prevent the undecidability of reasoning services. One

of the restrictions in OWL is the inability to use the same identifier (URI) for a class and an individual. To overcome this limitation, *punning* in OWL allows one to treat a class and an individual with a same URI as if they were different (Motik, Patel-Schneider et al. 2009).

Three subsets of OWL called *profiles* are used to create ontologies for a specific purpose by further restricting the use of constructs. These profiles are EL, QL and RL (Motik, Patel-Schneider et al. 2009).

EL is the best candidate for large ontologies, such as ontologies in the Life Sciences that contain complex descriptions based on the composition of entities, for example the chemical functional groups ontology described in Subsection 3.2. EL does not allow the use of negation, disjunction, or universal quantification on properties in a class expression, cardinality restrictions, enumerations with more than one individual, disjoint properties, irreflexive object properties, functional and inverse object properties, symmetric or asymmetric object properties. However, reasoning tasks for it can be performed in polynomial time with respect to the size of the ontologies (Motik, Grau et al. 2009). Reasoners that support EL include ELK (Kazakov, Krötzsch et al. 2011), CEL (Baader, Lutz et al. 2007) and CB (Kazakov 2009).

QL targets the creation of ontologies that can be easily integrated with RDBMSs in order to store the data in relational databases and access it via query rewriting. This profile disallows existential qualified restrictions, enumeration of individuals and literals, universal qualified restrictions, cardinality restrictions, disjunction, property chains, functional and inverse functional properties, transitive properties, keys (similar to

inverse functional properties), individual equality assertions and negative property assertions (Motik, Grau et al. 2009). Owlgres (Stocker and Smith 2008) and OWLIM (Kiryakov, Ognyanov et al. 2005) are reasoners that support QL.

Finally, RL is created for ontologies that require scalable reasoning while still being expressive in such a way that they can be integrated with rule-based technologies. This profile does not allow disjoint unions of classes and reflexive object properties (Motik, Grau et al. 2009). Reasoners that support RL include OWLIM, and Jena.

In addition to profiles, syntaxes for different purposes (*e.g.* human consumption) have been created for OWL. Manchester OWL syntax, a human readable one, is presented below.

**Manchester OWL Syntax**

In this document, we adopt the Manchester OWL syntax (Horridge and Patel-Schneider 2008) for presentation purposes to represent OWL axioms. We introduce a variant where labels[1] are identifiers for classes and properties.

For instance, with Manchester OWL syntax, the classes, properties and individuals of Figure 4 are represented in (8). The classes `Manager`, `Employee` and `Project` are preceded by the word *Class*, and the individuals `Joe` and `Mary`, `Apps` and `Migration` are preceded by the word *Individual*. The word *Types* is used to denote the classification of an individual as an instance of a class. A tabulator approach is used to distinguish to which entity the information belongs.

[1] Our ontologies contain a human readable label and description using the `rdfs:label` and `dc:description` annotations respectively.

```
Class: Manager
Class: Employee
Class: Project
Individual: Joe
  Types:
    Employee,
    Manager
    Facts:
   manages Apps
Individual: Mary
  Types:
    Employee                                                    (8)
  Facts:
  manages Migration
Individual: Apps
  Types:
    Project
Individual: Migration
  Types:
    Project
```

### 2.2.3 Reasoning

In previous sections we have used the word *infer* to denote the drawing of new knowledge based on the axioms and facts included in an ontology. This is one of the tasks that can be executed with a reasoner, hereby referred to as reasoning services.

Although some of the reasoning services of Description Logics have been shown as an intractable problem (*i.e.* requiring a huge amount of time for even modest input sizes, NEXPTIME-complete in the worst case (Horrocks and Schneider 2003)), this did not prevent their implementation in Description Logics reasoners using tableaux algorithms (Horrocks, Sattler et al. 1999), with good practical performance that rely on sophisticated optimization techniques.

Reasoners that provide services over OWL ontologies include Pellet (Evren, Bijan et al. 2007), Racer (Haarslev and Möller 2001), FACT++ (Tsarkov and Horrocks 2006) and HermiT (Shearer, Motik et al. 2008). The standard reasoning tasks provided by DL reasoners are: i) *consistency checking*, which ensures that an ontology does not contain contradictions, ii) *concept satisfiability*, that determines if a class can have at least one instance, *iii) classification* (also known as subsumption), which computes the 'is a' relation based on class descriptions to obtain an inferred hierarchy, and iv) *realization,* that classifies an individual into its most specific class (Donini, Lenzerini et al. 1996; Baader and Nutt 2003; Evren, Bijan et al. 2007). In practice, these services can be accessed using the command line or via the OWL API (Bechhofer, Noppens et al. 2007).

The constructs used in the creation of an ontology define its expressivity in terms of the Description Logics family it belongs to and the complexity of executing the reasoning services over the ontology (Baader 2003). The meaning of the letters composing the name of a description logics family follows. The letter $\mathcal{S}$ (also known as $\mathcal{ALC}$) indicates the use of basic constructs: concepts, complement of concepts, conjunction and disjunction of concepts, existential and universal quantified restrictions over roles, plus role transitivity (*e.g.* 'is related to' can be asserted as transitive). The letter $\mathcal{H}$ indicates the use of role hierarchy (*e.g.* 'manages' is can be a sub property of 'is related to'). The letter $\mathcal{R}$ refers to the use of complex role inclusions to express propagation of one property along another one, (*e.g.* manages o 'has part' -> manages indicates that the composition of manages and 'has part' can be seen as a subproperty of manages) (Horrocks, Kutz et al. 2006). $\mathcal{O}$ indicates the use of nominals to

define classes (*e.g.* {male, female} for Gender). $\mathcal{I}$ refers to the use of inverse role (*e.g.* manages can be the inverse of 'has manager'). $\mathcal{N}$ indicates the use of number restrictions (*e.g.* 'manages at least 1'), $\mathcal{Q}$ indicates the use of qualified cardinality restrictions (*e.g.* 'manages at least 1 Department'). Finally, $(\mathcal{D})$ refers to the use of data types (*e.g.* xsd:*string*). The Description Logics family $\mathcal{SROIQ}(\mathcal{D})$ is the family underlying OWL 2 (Horrocks, Kutz et al. 2006).

### 2.2.4 Semantic Web Rule Language

The Semantic Web Rule Language (SWRL) (Horrocks, Patel-Schneider et al. 2004) allows the representation of rules (*i.e.* implications between antecedent and consequent) using OWL constructs (such as classes, properties, individuals). SWRL rules can be combined with an OWL knowledge base and used by reasoners to infer new knowledge. SWRL uses rules in the form of an antecedent (body) and a consequent. If the body holds (is true) then the consequent must also hold. For example "If *X* manages Y and X is an instance of Employee then X is an instance of Manager" Here, *X* and *Y* represent any individual, "If *X* manages Y and X is an instance of Employee" is the antecedent and "X is an instance of Manager" is the consequent of the rule. Rules can be used by reasoners (*e.g.* Jess (Eriksson 2003), Pellet, HermiT) to infer new knowledge. For instance, to infer all instances of the Manager class.

Although SWRL is not yet a recommendation of the W3C, it is a member submission since 2004 and it is the supported language in ontology management applications such as Protégé (Noy, Crubezy et al. 2003). SWRL is compatible with the RIF rule exchange

standard. SWRL makes use of *safe rules* (Motik, Sattler et al. 2005) to maintain decidability. Safe rules contain certain restrictions to ensure decidability and improve the performance of current applications, *e.g.* all variables on the consequent must also occur in the antecedent and are considered to be universally quantified.

The SWRL's human readable syntax used by Protégé is used to denote rules in this manuscript.

### 2.2.5 Open World Assumption and Unique Name Assumption

Two important aspects of OWL semantics that influence reasoning services are *the Open World Assumption* (OWA) and the lack *of Unique Name Assumption* (UNA), both inherited from Description Logics.

The Unique Name Assumption (UNA) states that different names represent different individuals. Because this assumption is not used, it must be explicitly stated in an OWL ontology that two things are in fact the same or different.

The Open World Assumption (OWA) states that the truth value of a statement is unknown unless it is explicitly asserted or inferred to be true or false. DL reasoners implement negation by contradiction, *i.e.* they can only answer true or false on statements that can be proven (*i.e.* those that are known), with the exception of some reasoners, such as TrOWL (Thomas, Pan et al. 2010) and Racer that allow the reasoning with Closed World Assumption (CWA) which infers the value of a statement as false if it is unknown.

## 2.3 Ontologies and Relational Databases

(Villanueva-Rosales and Dumontier 2008) presents a preliminary report on the differences between ontologies and relational databases with respect to query answering. While relational databases are highly optimized for query answering, ontologies allow a more expressive formalization with the description of more complex relationships between entities than the ones allowed in relational databases. For instance, transitive relations are included in OWL and missing in (standard) relational databases. Although transitive relations can be *emulated* in relational databases (extending the relational model), this process is not straightforward, as they require recursive SQL queries that extend relational algebra, such as the implementation of hierarchical relations for instance. This is hard to maintain given the information needed *a priori* (*e.g.* database schema, data types) that may limit the scope of the application, making it domain dependent. Also, the user would need to have a previous training in SQL queries, which is uncommon among users of database applications.

Table 3 includes a summary of the representation of similar constructs in the entity-relationship model, the relational model and OWL 2. The entity-relationship model is the intended conceptualization when a relational model is created while the relational model contains the underlying logics behind relational databases as previously mentioned. It can be seen in this table that the relational model, entity-relationship model and ontologies share not only conceptual representations, but also semantic constructs. Notice that the accuracy of this comparison requires the relational model to be at least in Third Normal Form.

Table 3. Summary of the representation of similar constructs in the entity-relationship model (ER), relational model and OWL.

| **ER model** | **Relational model** | **OWL** |
|---|---|---|
| Entity | Relation | Class |
| Binary relation | Relation or attribute | Object property |
| N-ary relation, n>2 | Relation | Not available |
| Specialization/ Parthood relation (EER) | Relation and referential integrity constraint | Subclass hierarchy<br>Parthood hierarchy |
| Mapping of Union types (EER) | Attribute | Class union. |
| Instance | Tuple | Individuals |
| Attribute (simple, composite, multivalued) | Atomic attribute | Data property |
| Key attribute | Primary key | Inverse functional object property / URI. |
| Value set | Domain | Domain and range |
| Cardinality ratio | Keys | Cardinality constraints |
| Cross references | Referential integrity constraints, triggers and assertions | Cardinality and value constraints |

In summary, relational databases (Section 2.1) are created to efficiently store and query data, they are highly optimized towards this goal and they are not necessarily shared. Therefore, it can be assumed that all the data relevant to the domain of discourse is contained in the database (Closed World Assumption). It is also assumed that different database identifiers refer to different entities (Unique Name Assumption). The conceptualization of a relational database is described in the database schema, which contains the conceptualization of the domain of discourse. The language to describe a database schema in relational databases is SQL, however its syntax can vary depending

on the RDBMS being used. Further semantics or manipulation of data is normally encoded in the database application.

The goal of ontologies (Section 2.2.2) is to provide meaning according to the domain of discourse, a shared conceptualization that may not be complete and for which actual instances of the model are required to verify its validity. Therefore, it is fair to assume that the knowledge is not complete (Open World Assumption). Ontologies can be represented (in the Semantic Web) using RDF or OWL for which serialization syntaxes are available, such as Manchester OWL Syntax, and which are independent of the ontology management system being used. The manipulation of ontologies can be done via a Description Logics reasoner, which is also independent of the specific ontology or the ontology management system being used.

The following chapter describes in more detail how information, and in particular, information from databases can be formalized using the Semantic Web as well as the advantages of such a formalization.

# Chapter 3 Formal Knowledge Representation using Ontologies

Relational databases and ontologies can be seen as a different models for representing, storing and retrieving data, each created with different goals. Despite these differences, we believe that by combining these two models we can improve the way we manage data and use it to discover implicit knowledge. Our goal is to explicitly represent with ontologies the meaning behind a row or a foreign key in a relational database.

The first step towards this goal is to formalize the knowledge represented in the database using an ontology. The examples presented in this chapter are applications in the Life Sciences, but the features and challenges described are not restricted to this domain.

## 3.1 Formalizing Knowledge as Ontologies

When creating or reusing an ontology, we make an ontological commitment; we agree to use the terminology in a way that is consistent with the axioms and the documentation in the ontology (Gruber 1993). The same applies when we choose a language to represent an ontology. In general, the more expressive the vocabulary, the more we can restrict the domain being described. For instance, OWL allows us to

restrict the parts of a water molecule to be exactly two atoms of Hydrogen and exactly one atom of Oxygen, while RDF can only allow us to define the relation between a water molecule and Hydrogen or Oxygen. Moreover, the atoms of Oxygen and Hydrogen can be the same entity in RDF, given that disjointness cannot be represented.

The manual generation of expressive ontologies is tedious: many design decisions need to be made, which in the majority of cases are challenging and non trivial according to our experience in the manual design of Life Sciences ontologies and ontologies in the statistical domain (Villanueva-Rosales and Dumontier 2008; Dumontier and Villanueva-Rosales 2009; Dumontier, Ferres et al. 2010). Some of these decisions are listed below.

***Classes vs. instances.*** Is an entity better described as a class or as an instance? How does choosing one or the other influence the inferences obtained?

***Classes vs. properties.*** Is a phenomenon better modeled as a property or as a class? What are the advantages and disadvantages of each approach?

***Class restrictions.*** When should restrictions be added to a class? Is a restriction better modeled with a universal or an existential restriction? Does reasoning with class A require a closure axiom? Will cardinality constraints hold for class A?

***Properties.*** In order to describe a specific characteristic of an entity, is it better to use an object property, a data property or an annotation property? What are the advantages of using one or another? When is it appropriate to use a hierarchy of properties (with the OWL sub property constructor)? Is there an advantage to defining transitive, reflexive, symmetric, asymmetric and composed properties? Is it necessary to assert the domain and range for the property created?

***Hierarchies and Mereologies.*** What is the best way to represent the hierarchy of the biological entities being described? When is the 'part of' relation preferred (mereology) over the 'is a' relation (class hierarchy)? What about multiple parents?

***Ontology reuse.*** What are the advantages of extending available ontologies? Is the import of the whole ontology the only option available? What if the domain of discourse is not completely compatible with the ontology being imported? Is there another option to import only part of an ontology?

***Identifiers.*** If there are already identifiers available to describe an entity of interest, what are the advantages between using one or another? If no identifier is available, what is preferred to use, a URI, or a URN (Clark, Martin et al. 2004) such as LSID (Martin, Hohman et al. 2005)?

***Open World Assumption and Unique Name Assumption.*** These assumptions underlying OWL can influence ontology design decisions, since the explicit declaration of restrictions might be necessary to obtain the desired inference (*e.g.* negated axioms, cardinality restrictions).

***Spatial and temporal notions.*** How can the spatial location of an entity be represented? For example, how should relative *vs.* absolute locations be represented? What level of granularity should be used? What if a location changes over time? How do spatial locations relate to each other? How should we differentiate between characteristics that change over time and those that do not change?

The area of ontology engineering aims to provide semantic clarity, explicitness and to facilitate the reusability of represented information and knowledge (Soldatova and King

2007). Although some proposals in the area of ontology engineering (Uschold 1995; Noy and Hafner 1997; Guarino, Welty et al. 2000; Noy and McGuinness 2001), design patterns and workflows (Aranguren 2005), best practices (W3C 2004b), relation formalisms (Smith, Ceusters et al. 2005) and normalization of domain ontologies (Rector 2003) have been published, this is a relatively new research field and most of the methodologies and guidelines are not formalized or standard.

Most efforts in the generation of ontologies have been applied to the Life Sciences domain. Ontologies in this domain are often referred to as bio-ontologies. Three examples of manual design of bio-ontologies are presented as follows to provide an insight of the tasks and challenges involved in their design along with lessons learned.

## 3.2 Bio-ontologies

Hierarchies and classifications of living organisms have been manually created and applied in Life Sciences for the organization of organisms based on a common agreement, such as the NCBI taxonomy (Wheeler, Barrett et al. 2006). The Life Sciences can benefit enormously from the generation of ontologies that can automatically classify entities based on specific criteria. Consider for instance the area of Chemistry, where the structure of a chemical compound can define the chemical transformations it can participate in. The presence of functional groups in a chemical compound is key to describe the chemical reactivity of such a compound and compounds may be classified based on the presence of functional groups (Feldman, Dumontier et al. 2005). In (Villanueva-Rosales and Dumontier 2007a), we provide a logical description in OWL of

chemical structure such that it may be used to define functional groups for the purpose of compound classification.

In this chemical functional groups ontology, molecules have atoms as proper parts and atoms are connected to each other by a bond. Chemical bonds are represented using a symmetric property between atoms. Properties are also available to specify bond order *i.e.* 'has single bond with' or 'has double bond with' two atoms. While the most general bond property is 'has bond with', several sub-properties exist, such as 'has triple bond with', and 'has aromatic bond with'.

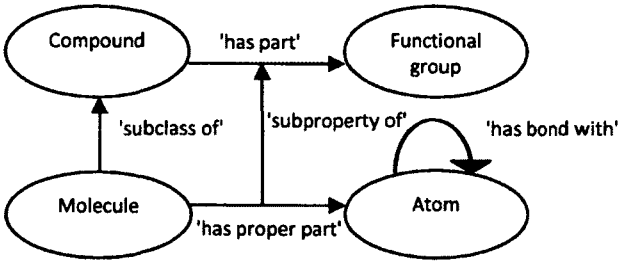



Figure 5. Ontology entities and their relationships in the chemical functional groups ontology (Villanueva-Rosales and Dumontier 2007a). Ellipses represent OWL classes while arrows represent OWL properties.

Functional groups consider composition and connectivity to define a specific chemical substructure. Specific organic compounds may be defined by virtue of the presence of specific functional groups. We define 35 chemical functional groups in OWL-DL by describing the necessary and sufficient atomic composition and connectivity (referred herein as the chemical substructure). For instance, the axioms in (9) represent the conditions for the hydroxyl functional group.

```
Class: 'Hydroxyl Group'
  EquivalentTo:
    'Carbon Group' and ('has single bond with' some
      ('OxygenAtom' and ('has bond with' some 'Hydrogen Atom')))
```
(9)

Similarly, alcohol is defined as an organic compound that contains the hydroxyl functional group as described in (10).

```
Class: Alcohol
  EquivalentTo:
    'Organic Compound'
     and ('has part' some 'Hydroxyl Group')
```
(10)

Therefore, molecules would be inferred to be an alcohol if they contain a set of atoms classified as a hydroxyl group.

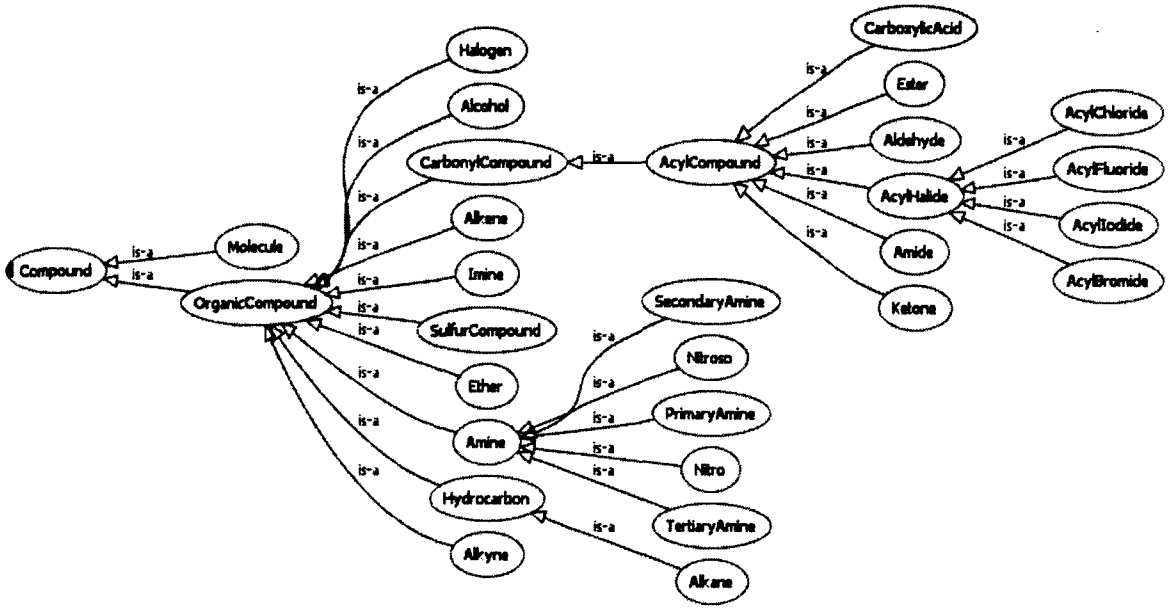



Figure 6. Fully inferred ontology hierarchy of organic compounds from (Villanueva-Rosales and Dumontier 2007a).

Given this definition, the hierarchy of organic compounds shown in Figure 6 is completely *inferred* (*i.e.* automatically generated by a reasoner). This hierarchy illustrates how complex definitions can be created from simpler ones but also opens the door to identifying all atoms that compose a functional group, which was not available

in previous work (Feldman, Dumontier et al. 2005). This allows a user to navigate at various levels of granularity, from an atom to an organic compound.

The expressivity of this ontology is $\mathcal{ALCHOIQ}$, containing atomic and complex classes, negation, intersection, existential and universal restrictions, property hierarchy, enumerated classes and qualified cardinality restrictions.

Despite the advantages of using ontologies, data management applications, particularly in the Life Sciences, rely on databases and only use ontologies as a vocabulary to annotate their data, *e.g.* PharmGKB (Hewett, Oliver et al. 2002).

The Open Biomedical Ontologies (OBO) is a shared portal of biological/medical ontologies that includes the popular Gene Ontology (GO) (Harris, Clark et al. 2006). OBO controlled vocabularies and taxonomies provide standardized vocabularies that are used in the annotation of biological information, which helps make information more accessible for computer interpretation. Through the OBO Foundry effort, OBO are being redesigned and mapped to the Basic Formal Ontology (BFO) (Grenon, Smith et al. 2004), a domain independent ontology that provides distinction between objects and processes and can be linked using basic relations (Smith, Ceusters et al. 2005). Together, they provide a powerful platform to annotate domain specific knowledge.

Unfortunately, not all of the ontologies previously described can be used for question answering, knowledge discovery, and model checking. For instance, despite the OBO foundry effort, OBO ontologies cannot be used in this way. A biologist would not be able to query a database about individuals that are proteins and catalyze reactions (also known as enzymes) using OBO ontologies. This drawback can be explained by many

factors including: i) use of non-standard non logic-based languages, ii) a lack of explicit logical descriptions to define class membership in terms of their properties iii) unrestricted use of classes and instances (this is only allowed in OWL Full, making the reasoning services undecidable) and iv) interchangeable use of 'is a' and 'part of' relations, among others. These issues are not exclusive of OBO ontologies, they are also shared by many of the current bio-ontologies and other domain ontologies.

With the increasing generation of bio-ontologies, guidelines for ontology design are also becoming more necessary to avoid the issues previously described

In the following subsections we present a methodology for the generation of ontologies based on our experience with manual generation of bio-ontologies and ontologies in the statistical domain (Villanueva-Rosales and Dumontier 2007a; Dumontier, Faizan et al. 2008; Dumontier, Ferres et al. 2008; Dumontier and Villanueva-Rosales 2008; Villanueva-Rosales and Dumontier 2008; Dumontier, Ferres et al. 2010). These ontologies are available at http://ontology.dumontierlab.com unless otherwise stated.

## 3.3 A Recipe for Ontology Design

For ontology design, we extend the methodology proposed in (Noy and McGuinness 2001) with a multi-layer ontology design of increasing expressivity proposed in (Dumontier and Villanueva-Rosales 2007). This ontology design process, introduced in (Villanueva-Rosales and Dumontier 2008), can be summarized in the following steps:

1. Define the scope and requirements (use cases) of the ontology.
2. Determine essential concepts, reusing existing ontologies where possible.

3. Construct a primitive hierarchy of concepts.
4. Map to an upper level ontology.
5. Assign relations between concepts and attributes.
6. Design a second layer of complex definitions that imposes additional logical restrictions and establishes necessary and/or necessary and sufficient conditions. If applicable, design a third layer of complex definitions for a specific application.
7. Repeat steps 2-6 until requirements are satisfied for semantic query answering over populated ontologies.

In each one of these steps an increasing ontological commitment is introduced due to the increase in the restrictions imposed on the description of the domain being formalized.

This methodology was used for designing the first version of yOWL (Villanueva-Rosales and Dumontier 2008), an ontology to model information found in the *Saccharomyces* Genome Database (SGD) (Cherry, Adler et al. 1998). Each one of the steps is listed below and illustrated using yOWL.

### 3.3.1 Define the scope and requirements (use cases) of the ontology

In this process, the domain of discourse along with the application of the ontology is described. A list of competence questions is recommended to verify that the ontology satisfies its requirements (Noy and McGuinness 2001). The scope of the yOWL ontology is to model the entities and their relations found in the SGD database, such that one

could effectively search this knowledge base using expected relations rather than with a collection of keywords. One of the questions to be answered by yOWL is: "Find genes that play a role in transcription and are participants in some genetic interaction experiment". SGD is a free and widely used resource for yeast genomic and proteomic information, in which structural and functional chromosome features (telomeres, genes, *etc.*), database cross references, molecular functions, cellular components, biological processes, interactions, pathways, phenotypes and literature references are described. The advantage of creating an ontology based on legacy data (*e.g.* the SGD database) is that one can test and evaluate the ontology using such a data for question answering.

### 3.3.2 Determine essential concepts, reusing existing ontologies when possible

Basic concepts that are the building blocks of the ontology are identified in this step. For yOWL, an initial set of concepts was collected to represent the types and attributes of the data located in the SGD data files. Concept definitions were obtained from the SGD glossary, WordNet (Fellbaum 1998) and Wikipedia (Wikipedia 2001) and were added to the classes using the `rdfs:comment` annotation property for human consumption.

To illustrate the conceptual process, we will describe the methodology for the *interactions* file. This file describes results from physical or genetic interaction experiments and each row contains columns for the pair of interacting genes/proteins, the type of interaction experiment, the impact on viability and a reference to a publication in which it was reported. To represent this information in the ontology, the following concepts were created: a) the class of `InteractionExperiment` and more

specific classes from an enumeration of types of interaction experiments (*e.g.* SyntheticLethalityExperiment, AffinityCapture-MS, *etc.*). Values captured in the *viability* column were represented as attributes of the entity Experiment: viability states (viable, non-viable) and growth states (slow growth, *etc.*). References to journal articles identified by PubMed identifier are represented by a Publication class.

Similarly, data found in the *go_slim_mapping* file corresponds to information annotated with Gene Ontology terms. However, at the time of designing yOWL, no OWL document was available for this subset of GO. Thus, the GO Slim OWL ontology (Villanueva-Rosales and Dumontier 2007b) was constructed with concepts that are equivalent (via owl:EquivalentClass) to those in the complete OWL GO ontology (Gene Ontology Consortium 1999), but with English class names like "Nucleus" rather than alphanumeric class names favoured by OBO ontologies (*i.e.* GO_0005634).

### 3.3.3 Construct a primitive hierarchy of concepts

Subsumption of classes is identified in this step. A hierarchy of concepts for the yOWL primitive ontology (yowl-primitive) was developed by iteratively categorizing the set of classes. For instance, a SyntheticLethalityExperiment is a type of SyntheticGeneticExperiment, which is a type of GeneticInteractionExperiment, which is a type of InteractionExperiment, which is a type of Experiment. The result is the design of a taxonomic branch that is homogenous, with respect to the property that specializes each branch, and increasingly specialized in that each child term can be easily differentiated from its parent by means of the values or constraints imposed on such a

property. In line with general ontology normalization techniques, all ontological terms in yOWL are asserted to have but a single parent.

### 3.3.4 Map to an upper level/integrated ontology

Upper level ontologies promise increased semantic coherency by helping to identify the basic types of domain entities and imposing restrictions on the relationships that these entities may hold. An upper level ontology should be carefully selected according to the purpose and scope of the ontology designed. The Basic Formal Ontology (BFO) provides a simplified framework to distinguish between qualities, objects and processes. Classes defined in the yOWL ontology are mapped to the BFO ontology. For example, 'Validation status' is a type of Quality, a Publication is a type of Object and an Experiment is a type of Process. In doing so, we anticipate that our classes (and the restrictions placed on them) will be compatible with second generation OBO ontologies and other ontologies that also adopt the BFO framework. An important criterion in BFO is that all classes in an ontology are actually instantiated in reality. By committing to this criterion, we are committing to instantiate each one of the classes in our ontology. When reusing an upper level ontology, there is an additional ontological commitment to the definitions and restrictions described in such an ontology.

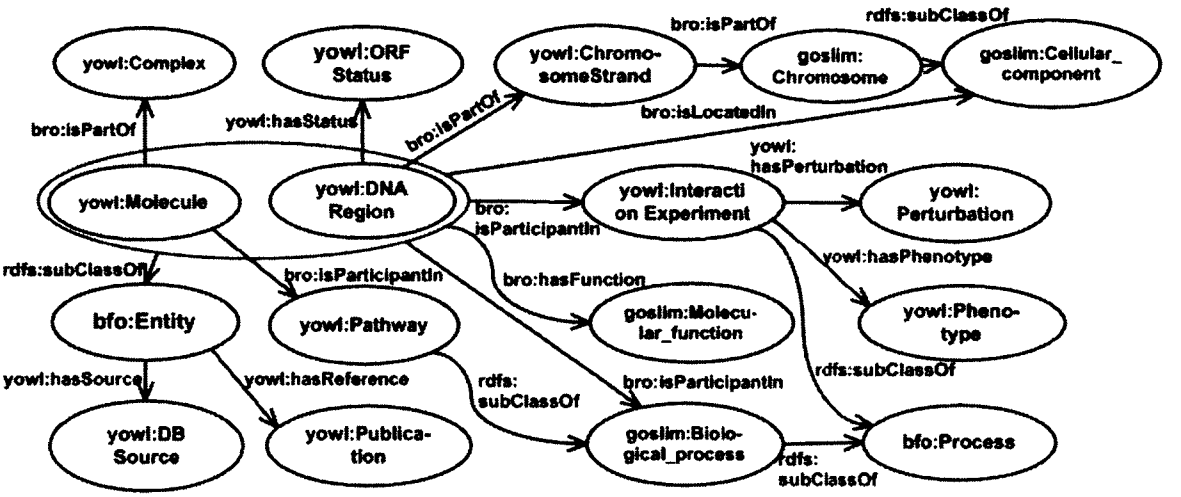
### 3.3.5 Assign relations between concepts and attributes

An essential part of the modeling process is to establish which relations exist between a set of concepts. To do this, we need two things: 1) a set of basic relations to draw from and 2) the description of the concepts. yOWL reuses the Basic Relation Ontology (BRO)

(Dumontier 2007b). This ontology hierarchically organizes basic object relations (Grenon, Smith et al. 2004; Smith, Ceusters et al. 2005) for use in OWL ontologies. A root relation 'is related to' provides a generic relationship between any two entities and also provides object-process, object-quality, mereological, spatial and temporal relations. The New Upper Level ontology (NULO) created at Dumontier Lab (Dumontier 2007a), maps the domain and ranges of BRO relations with BFO concepts so as to semantically constrain the assignment of relations and provide enhanced reasoning and inference capabilities. Taken together, NULO consists of 36 classes, 50 object properties, 2 data properties and 17 annotation properties. Similar to the use of upper level ontologies, the use of NULO in several ontologies imposes extra restrictions that generates further inferences and may facilitate semantic integration in an abstract level. New object properties were added to describe the more specific relations required (but not restricted to) in this domain.

Several data properties were also introduced to relate information that is intrinsic to a specific entity. For instance, an experiment may be related with the publication where it is described using the data property 'has citation'.

Figure 7 contains a graphical representation of the yOWL ontology.




yOWL

**Namespaces**
xsd: http://www.w3.org/2001/XMLSchema
rdfs: http://www.w3.org/2000/01/rdf-schema
owl: http://www.w3.org/2002/07/owl
dc: http://purl.org/dc/elements/1.1/
bfo: http://www.ifomis.org/bfo/1.0.1
bro: http://ontology.dumontierlab.com/bro
goslim: http://ontology.dumontierlab.com/goslim-2.0
yowl: http://ontology.dumontierlab.com/yowl-primitive

Figure 7. Selected classes and relations from yOWL ontology. Classes are represented by ellipses and properties are represented by arrows. Modified from (Villanueva-Rosales and Dumontier 2008).

### 3.3.6 Design a second (and optionally a third) layer of complex definitions that imposes additional logical restrictions and establishes necessary and/or necessary and sufficient conditions

Necessary and sufficient conditions that hold in the domain of discourse are asserted in this step of the methodology.

The yOWL primitive hierarchy of concepts was augmented with necessary and sufficient conditions to ensure the proper classification of instances. Necessary conditions were added according to definitions of genome structure and function *(i.e.* a chromosome strand is a single stranded DNA molecule that is part of a chromosome). Necessary and

sufficient conditions were added for single property varying entities (*i.e.* a dosage rescue experiment is a dosage interaction experiment that has a viable outcome) or to represent a value partition (*i.e.* the quality of viability is defined by a state of viable or non-viable). The yOWL complex ontology (yowl-complex) is comprised of 257 classes, 60 object properties, 12 data properties and 22 annotation properties. These include the BFO and GOSLIM ontologies. It belongs to the $\mathcal{SHIF}(\mathcal{D})$ description logics family.

Once an ontology has been designed, the following step (not always required) is to populate such an ontology.

## 3.4 Ontology Population

In the ontology population phase, *instances* of the ontology classes are created. This phase is crucial for question answering. If data integration is required, statements made in different sources about a single resource should be merged.

For the yOWL ontology, the ontology population involved: i) assigning names, ii) asserting class membership, and iii) determining proper relations between entities. In the assignment of unique names, Universal Resource Identifiers (URIs) were constructed from an assigned base namespace based on its original source, such as PubMed (National Centre for Biotechnology Information (NCBI)), plus a unique identifier. URIs were created for information mapped from database providers who do not make their information available in RDF and for which a namespace was not available. For instance, the PubMed identifier for the yOWL paper is 18562252 and the corresponding identifier for this publication is urh:lsid:ncbi:nlm:nih.gov:pubmed:18562252. When information

about different identifiers referring to the same entity was available, such as cross-references tables, the identifiers were related with the OWL sameAs property to merge the information related with those identifiers as if they were the same identifier. In the assignment of types, the yOWL ontology sufficiently covers the genomic and proteomic data obtained from SGD tab files. However, given that there was no relational database schema available from which we could determine the composition of the tables that generated the tab files, the specific relations between the data had to be manually determined. The data contained in SGD records was mapped to an OWL representation with in-house developed parsers writing in the PHP scripting language (Lerdorf, Tatroe et al. 2006) whose input was the SGD tab files and output an OWL file.

## 3.5 Question Answering

The design of reasoning capable applications can facilitate information retrieval and knowledge discovery about a subject of interest. We demonstrate, by means of examples, how a scientist could execute question answering over the yOWL ontology, retrieve information from yOWL at various levels of ontological granularity and query across data having multiple synonymous identifiers. These questions were created as competency questions to validate that yOWL satisfies the requirements for which it was created. Some of these questions, for which an answer is correctly retrieved by the yOWL ontology, include:

1. Find all individuals that have a molecular function.
2. Find pathway participants that are also physical interaction participants.

3. Find genes that play a role in transcription and are participants in some genetic interaction experiment. Return both the genes and their associated publications.
4. Find all information related to Gene NSA3.

Note that query 4 is also used as an example in the introductory chapter. This query retrieves all the individuals NSA3 is related to at the most general level of granularity: EBI, CGD, NCBI, GenBank/EMBL/DDBJ, BioGRID, DIP and CandidaDB (database sources), verified (ORFStatus), chromosome8_Watson, proteasome_complex, GO_S000001094_Ribosome_biogenesis_and_assembly, GO_S000001094_Nucleolus, GO_S000001094_Protein_catabolic_process and GO_S000001094_Protein_binding (GO identifiers). It is also related to a large set of interactions, and a set of publications including PMID:16922378. This exploratory query can later be refined searching for more specific types of relations between NSA3 and other individuals (*e.g.* find all the molecular functions related with NSA3 or the location of NSA3).

## 3.6 Knowledge Discovery

To further illustrate how implicit knowledge can be made explicit using ontologies and reasoning services we will use as example the Pharmacogenomics ontology described in (Dumontier and Villanueva-Rosales 2009) which models a domain that uses data from different sources and originally generated for different purposes. Pharmacogenomics aims to understand the response to a drug treatment with respect to genetic variation and is relevant to Personalized Medicine (Evans and Relling 2004). This ontology was

created following the methodology described in Section 3.3 to represent the information stored in the Pharmacogenomics and Pharmacogenetics knowledge base (PharmGKB) (Hewett, Oliver et al. 2002). Some of the competency questions considered for the ontology design were: What is the most effective drug treatment for an individual with a given genetic profile that suffers from a particular disease? Which drugs yield side effects?

Figure 8 shows the main classes and properties in the Pharmacogenomics ontology, where the key class is DrugTreatment. This ontology makes use of the composite roles hasParticipant o hasPart -> hasParticipant which generates the necessary inference that a drug and gene that are participants in a drug gene interaction are also participants in the drug treatment, if the interaction is part of the drug treatment. Also notice the use of classes for n-ary relations, relations with more than two participants that cannot be represented by an object property, such as DrugTreatment that has as participants instances of the classes 'Biological Measure', Disease, Drug and Gene. The Pharmacogenomics ontology (pharmgkb) contains over 40 classes, 10 object properties and 5 data properties. In addition, this ontology reuses the biomedical measure ontology (biomedical-measure-primitive), which contains 123 classes on a variety of measures including those that are biophysical, clinical, cognitive, genotyping, metabolic, pharmacokinetic, pharmacodynamic and taxonomic.

The Pharmacogenomics ontology was populated using PharmGKB web services with a total of 1700 individuals (genes, diseases, drugs) annotated with a pharmacogenomics term. This ontology was extended to generate the Pharmacogenomics of Depression

ontology (pharmacogenomics-depression) which includes information about entities manually extracted by domain experts from over 40 publications, many of which were initially identified by PharmGKB curators, and augmented with more recent literature. As of January 1, 2008 the ontology contained statements about depression from 11 publications involving 45 genes / gene variants, 57 drugs annotated with 19 classes of antidepressants, 45 drug treatments, 47 drug-gene interactions, 29 clinical outcomes, 10 drug-induced side-effects, and 8 gene-disease interactions. To illustrate the utility of the Pharmacogenomics of Depression knowledge base, we describe an application scenario: A psychiatrist diagnoses an elderly patient with depression based on the results of the Hamilton depression method. Considering the physical condition of the patient, postural hypotension must be avoided (a common side effect of many antidepressants). A query that asks for all drugs that have been used to treat depression (pgkb:PA447278) with a postural hypotension side effect of less than 5% of cases is described as follows (in Manchester OWL Syntax):

```
'drug' that 'is participant in' some
('drug treatment' that 'is related to' value 'Postural Hypotension' and                (11)
'is related to' some ('side effect rate' that
'has value' some double[<5] and hasUnit value percent))
```

Of the six drugs that are used to treat depression included in the ontology (Amitriptyline, Citalopram, Desipramine,Fluoxetine, Nortriptyline and Venlafaxine), only Nortriptyline (pgkb:PA450657) exhibits no side effect for postural hypertension. To answer the question, the reasoner invokes reasoning over a number of expressive elements of the ontology: (a) the bro:hasParticipant role chain *infers* drugs involved in

a drug treatment when this information is actually specified as part of a drug–gene interaction in which the drug is involved and (b) 'is related to' is a super property of all properties including 'has participant', which is used to assert the relationship between drug treatments and diseases as well as drug treatments and side effect rates. This ontology contains information about diseases, drugs, drug treatments, and interactions from PharmGKB, but also reuses other ontologies spanning the domain of biological measures and contains manually curated data. As a result, data can be retrieved seamlessly with no manual integration required. Based on this example, we expect that with the increasing of information available in RDF and OWL ontologies, knowledge discovery not only in the domain of discourse, but also spanning various domains of knowledge will become increasingly possible.

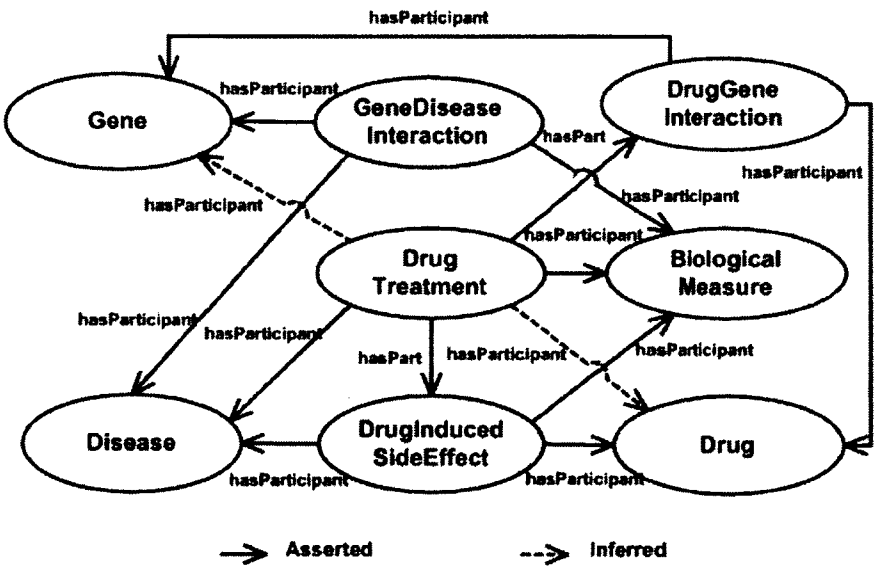



Figure 8. The pharmacogenomics ontology, built around the drug treatment which is composed in part by drug-gene interactions and drug-induced side effects (Dumontier and Villanueva-Rosales 2009).

## 3.7 Discussion

Although ontologies and databases were created for different purposes, by formalizing the data stored in relational databases using ontologies we can improve the way the data is represented by adding machine understandable knowledge that can facilitate question answering.

Question answering requiring data integration and discovery at various levels of granularity can be enabled as illustrated with the three bio-ontologies presented in this chapter. Question answering opens the door to new possibilities in how researchers manage not only their data, but also an increasing amount of knowledge.

A methodology to guide the generation of ontologies was proposed. However, many questions need to be answered when creating ontologies to guide design decisions. Ontology design decisions will influence not only the availability of reasoning services, but also their performance. The rationale to choose the best alternative should consider not only the formal logic underlying the ontology language but also the characteristics of the entities being described and the requirements that the ontology being designed should satisfy. When populating ontologies using data from databases, the model should also consider how the data is represented in the source. Moreover, in-house and customized parsers need to be created to *mash up* relational databases' data in order to serialize it as ontology individuals and relations. Most of the time, these parsers cannot be reused and will need to be updated when the source is updated. These are some of the reasons that make the manual generation of ontologies a time consuming task.

In this thesis, we propose a framework to *automatically* formalize the content of databases by means of ontologies. The following chapter describes this framework.

# Chapter 4 Mapping Relational Databases to OWL Ontologies

In the previous chapters, the differences and similarities between relational databases and ontologies were presented. It was also illustrated that formalizing the content of relational databases as OWL ontologies can be used for consistency checking, realization and question answering with benefits that span multiple domains including Life Sciences.

In this chapter, how DBOwlizer *automatically* generates OWL ontologies from relational databases is described. A key element in this process is to identify patterns representing the type of information contained in each one of the targeted database tables. For example, does the table contain information about an entity such as a person or an employee? Does the table contain information about how two or more entities are related, such as the department for which an employee works? Does it describe specific types of entities, such as a manager who is also an employee? Such questions may be answered by examining the table composition. If good practices in design and normalization procedures were followed when creating the relational database (as described in Subsection 2.1.7), standard patterns can be found in the composition of the

tables given their primary key, foreign keys and other constraints imposed on the values a table can hold.

We represent the composition of a database schema with the relational-model ontology (section 4.2). This ontology can be used to classify the database schema elements given their composition in order to identify patterns used to represent the domain being modelled. Upon this classification we provide a set of rules to automatically generate a mapping from each database component to an OWL element. These heuristics are represented in the relational-to-ontology-mapping ontology (section 4.3.). Finally, we automatically execute the relational database to OWL mappings to generate an ontology that specifies the semantics of the domain being described in the database (Chapter 5). The ontologies used in the DBOwlizer framework are available on the project website: http://semanticscience.org/projects/dbowlizer/.

To illustrate the use of the ontologies described above and the database features analyzed by DBOwlizer's default heuristics we provide as a test case a database with information about employees. This database was created using good design practices and it is normalized in Third Normal Form (3NF). The tables of the employees database have different compositions in order to exemplify different mappings. Note that the name of the tables and columns in the database is irrelevant for DBOwlizer's heuristics, but this example is widely used in database courses. More information about this database is provided in the following section.

## 4.1 Test Case: The Employees Relational Database

Our test case involves a database concerning employees and their roles, contracts, departments, etc. The employees database contains fifteen tables (Figure 9) and ten views.

The first input to DBOwlizer is the database schema or the relational database to be mapped into OWL. As explained in Subsection 2.1.1, the database schema specifies the composition of the database using the variation of the SQL language supported by the RDBMS, which in this case is MySQL. To illustrate the statements found in the employees database schema and how this information will be represented and manipulated in the DBOwlizer framework, consider the statement CREATE TABLE for the table **manager** in the employees database in (12):

```
CREATE TABLE `employees`.`manager` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `entry_date` datetime NOT NULL,
  `project` int(10) unsigned NOT NULL,
  `time_assigned` enum('full-time','part-time') DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `FK_manager_proj` (`Project`),
  CONSTRAINT `FK_manager_emp` FOREIGN KEY (`id`) REFERENCES `employee` (`id`),
  CONSTRAINT `FK_manager_proj` FOREIGN KEY (`project`) REFERENCES `project` (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=latin1;
```

(12)

This table contains the columns **manager.id, manager.entry_date, manager.project and manager.time_assigned**. The column **manager.time_assigned** has the **NULL** value as default and an enumeration domain, which means that the only values allowed for that column are: **"full-time", "part-time"** and **NULL**.

The manager table contains a primary key and two foreign keys (keys are described in subsection 2.1.4). Integrity constraints imposed on a table are critical for DBOwlizer's heuristics.

The queries embedded in **views** are used by the DBOwlizer framework as a source of semantics of the domain described. For example, the **high_salary_employee** defined in (13) is composed by records of the table **employee** with a value equal or greater than 80,000 in the **salary** column. This view represents a *subset* of employees that are of particular interest of a user.

```
CREATE VIEW `high_salary_employee` AS
select `employee`.`id` AS `id`,                                    (13)
`employee`.`name` AS `name`,
`employee`.`address` AS `address`,
`employee`.`salary` AS `salary`
from `employee`
where (`employee`.`salary` > 80,000)
```

The elements of the employees database schema previously described are transformed into instances of ontology classes in the relational-model ontology. This ontology is described in the following section.

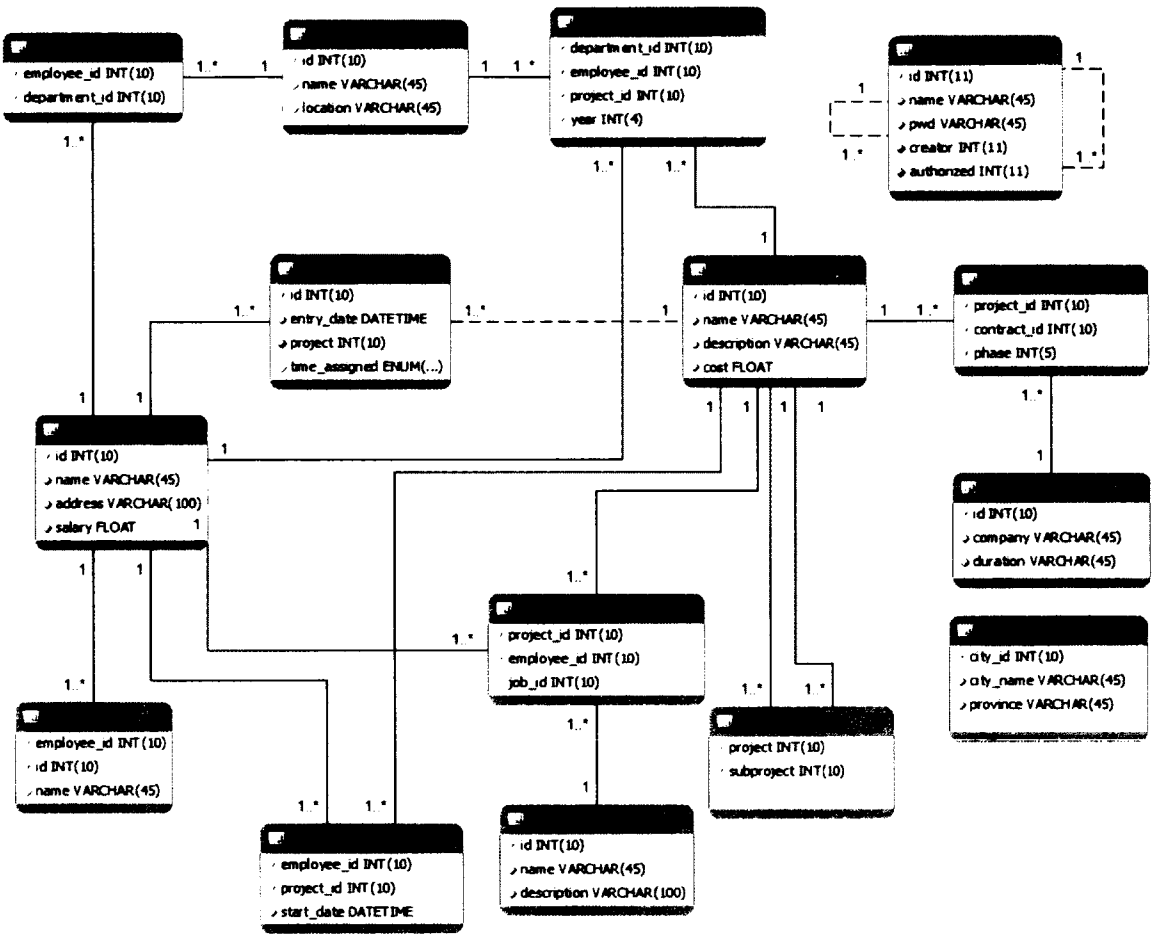



Figure 9. Employees database schema diagram (excluding views) auto-generated by MySQL Workbench ver. 5.1.18 (Zinner, Kojima et al. 2011). Tables are represented by rectangles. Lines denote foreign keys, if the referring column is part of the primary key the line is solid, otherwise it is dashed. Cardinality constraints are denoted by numbers in the lines. A diamond icon in blue is used for columns with not-null restrictions and in red for columns composing a foreign key. Columns composing the primary key are denoted by yellow key icons.

## 4.2 The Relational Model Ontology

Elements of the relational model such as relations, attributes and keys are described in the `relational-model` ontology, which is instantiated by the components of a database schema. This ontology was designed with the granularity required to classify tables based on their composition. This classification is crucial in DBOwlizer's heuristics for the

automatic generation of mappings in DBOwlizer and other methodologies described in related work (Chapter 6).

The relational-model ontology, like all the ontologies in DBOwlizer, was designed following the methodology described in Section 3.3, which includes a multi-layer approach (Dumontier and Villanueva-Rosales 2007) successfully applied in the biological (Villanueva-Rosales and Dumontier 2008) and the statistical (Dumontier, Ferres et al. 2008) domains. Following this multi-layer design in the relational-model ontology the primitive layer, named relational-model-primitive ontology contains a simple hierarchy of classes and a set of properties describing the relational model. The complex layer, named relational-model-complex ontology, contains restrictions that define different patterns in the composition of tables. Individuals in the DBOwlizer ontologies instantiate the complex layer classes and are separated in a different file whose name contains the postfix *individuals* (at the end of the name), *e.g.* employees-relational-model-individuals. Given that the individuals are a representation of the database schema and its records, the ontology containing the individuals is more likely to change than the other ontologies, and therefore is more convenient to have a separate ontology to update. Following the design methodology, classes and properties in the relational-model ontology are mapped to the Semanticscience Integrated Ontology (SIO) (Dumontier 2011), an ontology that provides a vocabulary for knowledge representation across physical, processual and informational entities.

The relational-model-complex ontology imports the relational-model-primitive ontology. It is composed of 55 classes, 15 object properties, 4 data properties and

corresponds to an $\mathcal{ALCHOI}$ description logic. The relational-model-individuals ontology contains 8 individuals representing datatypes such as db_string (which represents the string datatype) and restriction values such as db_not_null (which represents the restriction **NOT NULL**). These individuals are necessary for the classification of columns further explained in subsection 4.2.2.

The elements of the employee database described in section 4.1 are mapped to individuals in the employees-relational-model-individuals ontology. These 167 individuals instantiate classes of the relational-model-complex ontology. The process of mapping the database schema into OWL individuals is further explained in subsection 5.1.1.

Because the relational-model-complex ontology is built around the composition of the elements of the database schema, the object property 'has part' is widely used. This property relates a whole to its parts, and it is asserted as a subproperty of SIO's 'has part' object property. Unlike SIO's 'has part' property, it is not transitive in our ontology given that it is used in closure axioms and thus transitivity would introduce an inconsistency.

### 4.2.1 Database relation class

The relational-model ontology is built around the 'Database relation' class, whose members are relations and more specifically, the tables of a database. A table is composed by columns, formally known as attributes in the relational model, and integrity constraints represented in keys. These requirements for an individual to be

classified as a 'Database relation' are reflected in the necessary conditions of the class specified as follows:

```
Class: 'Database relation'
SubClassOf:                                                    (14)
('has part' some 'Database attribute') and
('has part' min 0 'Database key')
```

Database tables are mapped to instances of the 'Database relation' class. For instance, the table **manager**, described in the SQL statement in (12) will be represented with the OWL statements in (15). The relational-model ontology explicitly represents components of the tables, namely columns and keys.

```
Individual: empdb:Manager
  Types:
    'Database relation'
  Facts:
   dl:hasPart empdb:Manager.entry_date,
   dl:hasPart empdb:Manager.fk_manager_emp.foreign_key,
   dl:hasPart empdb:Manager.id,                                (15)
   dl:hasPart empdb:Manager.time_assigned,
   dl:hasPart empdb:Manager.fk_manager_proj.foreign_key,
   dl:hasPart empdb:Manager.project,
   dl:hasPart empdb:Manager.primary_key
```

Specific types of relations (tables) are classified based on the keys and columns that compose them. The name of the classes reflect their composition. This classification is based on the guidelines to represent entities and relationships in the relational model (subsection 2.1.8). Tables are automatically inferred as members of these specific classes with the aid of a DL reasoner. For instance, tables that represent a binary relation between two entities are classified as members of the 'Database binary relationship relation' class. The definition of this class includes tables that have a

'Database binary dependent primary key'. For example, the individuals Outsourceworkers and Worksfor representing the tables **outsourceworkers** and **worksfor** respectively are inferred as instances of this class.

The classification of the tables in the employee database is shown in Table 4. Note that the namespace in the individuals URI is not included for presentation purposes. This classification covers most of the characteristics used as preconditions in DBOwlizer's default heuristics, for the creation of relational databases to ontology mappings, and heuristics of related work, surveyed and compared in Chapter 6.

Table 4. Classification of the employee database tables based on their primary key composition in the relational-model ontology.

| Relation | Individuals |
|---|---|
| 'Database entity relation' | City, Contract, Department, Employee, Job, Project, Users |
| 'Database unary relationship relation' | Manager, Subproject |
| 'Database partially unary relationship relation' | Dependent |
| 'Database binary relationship relation' | Outsourceworkers, Worksfor |
| 'Database partially binary relationship relation' | Outsourceproject |
| 'Database n-ary relationship relation' | Projectjobemployee |
| 'Database partially n-ary relationship relation' | Temporalassignment |

### 4.2.2 Database attributes

Table columns are part of the building blocks in our representation of database tables. Columns are instances of the 'Database attribute' class. For each column, the datatype, default values and attribute restrictions are specified. For instance, the column **manager.time_assigned,** defined in (12), is asserted as a member of the class 'Database attribute'. The values of this column are restricted to an enumeration set

through the 'has value' functional data property. In addition to the allowed values "FULL-TIME" and "PART-TIME" defined in (12), a default value is added to the enumeration set, in this case the "NULL" value. The default value for this column is also specified in a separate statement. The complete description of the column **manager.time_assigned** in the relational-model ontology is specified in (16).

```
Individual: empdb:Manager.time_assigned
   Types:
    'Database attribute',
    'Database non-key attribute',
    'has value' some {"FULL-TIME"^^xsd:string, "PART-TIME"^^xsd:string,
"NULL"^^rdfs:Literal}
   Facts:
    'has default value' "NULL"^^xsd:Literal
```
(16)

Note that the datatype of "NULL" is asserted to be rdfs:Literal which corresponds to the range of the property 'has default value'. This built-in datatype denotes data values of any datatypes (Motik, Patel-Schneider et al. 2009) and therefore can accommodate any possible value that the property 'has default value' can have in different columns.

More specific types of attributes are described based on the restrictions imposed on the attribute, such as the values it can hold or the database constraints in which the attribute participates. For instance, 'Database foreign key attribute' is defined as attributes that compose a foreign key. A 'Database not-null attribute' is a 'Database attribute' that cannot hold "NULL" values, represented with the condition (in OWL): 'has attribute restriction' some dl:db_not_null. Table 5 lists the individuals representing columns of the employees database and their classification as

specific types of attributes in the relational model ontology. Again, the namespace in the individuals URI is not included for presentation purposes.

Table 5. Classification of the employee database columns based on the keys they compose and the values they can hold in the relational-model ontology.

| Database attribute | Attributes (columns) individuals |
|---|---|
| 'Database foreign key attribute' | Dependent.employee_id, Manager.id, Manager.project, Outsourceproject.contract_id, Outsourceproject.project_id, Outsourceworkers.employee_id, Outsourceworkers.project_id, Projectjobemployee.employee_id, Subproject.project, Projectjobemployee.project_id, Projectjobemployee.job_id, Subproject.subproject, Temporalassignment.department_id, Temporalassignment.employee_id, Temporalassignment.project_id, Users.authorized, Users.creator, Worksfor.department_id, Worksfor.employee_id |
| 'Database primary and foreign key attribute' | Dependent.employee_id, Manager.id, Outsourceproject.contract_id, Outsourceproject.project_id, Outsourceworkers.employee_id, Outsourceworkers.project_id, Projectjobemployee.employee_id, Subproject.project, Projectjobemployee.project_id, Projectjobemployee.job_id, Subproject.subproject, Temporalassignment.department_id, Temporalassignment.employee_id, Temporalassignment.project_id, Worksfor.department_id, Worksfor.employee_id |
| 'Database non-primary and foreign key attribute' | Manager.project, Users.authorized, Users.creator |
| 'Database candidate key attribute' | Employee.name |
| 'Database primary key attribute' | City.city_id, Contract.id, Department.id, Dependent.id, Dependent.employee_id, Employee.id, Job.id, Manager.id, Outsourceproject.contract_id, Outsourceproject.phase, Outsourceproject.project_id, Outsourceworkers.employee_id, Outsourceworkers.project_id, Project.id, Projectjobemployee.employee_id, Subproject.project, Projectjobemployee.project_id, Projectjobemployee.job_id, Subproject.subproject, Temporalassignment.department_id, Temporalassignment.employee_id, Temporalassignment.project_id, Temporalassignment.year, Users.id, Worksfor.department_id, Worksfor.employee_id |
| 'Database primary and non-foreign key attribute' | City.city_id, Contract.id, Department.id, Dependent.id, Employee.id, Job.id, Outsourceproject.phase, Project.id, Temporalassignment.year, Users.id |
| 'Database non-key attribute' | City.city_name, City.province, Contract.company, Contract.duration, Department.location, Department.name, Dependent.name, Employee.address, Employee.salary, Job.description, Job.name, Manager.entry_date, Manager.time_assigned, Outsourceworkers.start_date, Project.cost, Project.description, Project.name, Users.name, Users.pwd |

| | |
|---|---|
| 'Database non-foreign key attribute' | City.city_id, City.city_name, City.province, Contract.company, Contract.duration, Contract.id, Department.id, Department.location, Department.name, Dependent.id, Dependent.name, Employee.address, Employee.id, Employee.name, Employee.salary, Job.description, Job.name, Manager.entry_date, Manager.time_assigned, Outsourceproject.phase, Outsourceworkers.start_date, Project.cost, Project.description, Project.name, Project.id, Temporalassignment.year, Users.id,Users.name, Users.pwd |
| 'Database non-primary key attribute' | City.city_name, City.province, Contract.company, Contract.duration, Department.location, Department.name, Dependent.name, Employee.address, Employee.name, Employee.salary, Job.description, Job.name, Manager.entry_date, Manager.project, Manager.time_assigned, Outsourceworkers.start_date, Project.cost, Project.description, Project.name, Users.name, Users.pwd, Users.authorized, Users.creator, |
| 'Database not-null attribute' | City.city_id, City.city_name, City.province, Contract.company, Contract.duration, Contract.id, Department.id, Dependent.employee_id, Dependent. id, Employee.address, Employee.id, Employee.name, Employee.salary, Job.description, Job.id, Job.name, Manager.entry_date, Manager.id, Manager.project, Outsourceproject.contract_id, Outsourceproject.phase, Outsourceproject.project_id, Outsourceworkers.employee_id, Outsourceworkers.project_id, Outsourceworkers.start_date, Project.cost, Project.description, Project.name, Project.id, Projectjobemployee.employee_id, Subproject.project, Projectjobemployee.project_id, Projectjobemployee.job_id, Subproject.subproject, Temporalassignment.department_id, Worksfor.employee_id Temporalassignment.employee_id, Temporalassignment.project_id, Temporalassignment.year, Users.name, Users.pwd, Users.authorized, Users.creator, Users.id, Worksfor.department_id, |

### 4.2.3 Database keys

Another important element composing a table is a key (refer to Section 2.1.4 for more information about keys). Three types of keys are considered in DBOwlizer's heuristics: primary keys, foreign keys and candidate keys. These specific types of keys are represented by the classes 'Database primary key', 'Database foreign key' and

'Database candidate key' respectively, which are subclasses of 'Database key'. For example, the individual Manager.primary_key in (15) is an instance of 'Database primary key'.

Instances of the class 'Database key' are necessarily composed of one or more 'Database attribute'. In addition, a 'Database foreign key' references a 'Database primary key' or 'Database candidate key'. For example, the foreign key `FK_manager_proj` in the **manager** table is composed of the **project** column and references the **project.id** column (see (12)). The OWL statements representing the composition of this foreign key are:

```
Individual: empdb:Manager.fk_manager_proj.foreign_key
  Types:
   'Database non-primary foreign key',
   'Database foreign key'
  Facts:                                                    (17)
   references empdb:Project.primary_key,
   'has part' empdb:Manager.project
```

Following our ontology design criteria, more specific types of keys are defined using their composition. For instance, a 'Database Non-Primary Foreign Key' is a foreign key composed of attributes that are not part of the primary key. The foreign key in the **manager** table described in (17) satisfies these restrictions and therefore is asserted as an instance of this class. Likewise, primary keys are classified by analyzing the attributes that compose them. For instance, a 'Database binary dependent primary key' is composed *only* by attributes that are also part of two foreign keys referencing two different tables. An example of a member of this class is the primary key of the table

**worksfor** of the employees database (Figure 9). When a primary key contains at least one non-foreign key attribute, it is inferred as an instance of a class that contains the *partially* word in its name. For instance, empdb:Outsourceproject.primary_key, the individual that represents the primary key of the table **outsourceproject** is composed by the attribute Outsourceproject.phase (which is not part of a foreign key) and the attributes Outsourceproject.contract_id and Outsouceproject.project_id which are part of two foreign keys referencing Contract and Project respectively. Therefore, Outsourceproject.primary_key is classified as a 'Database partially binary dependent primary key'. Finally, depending on the number of foreign keys composing the primary key, such a primary key is classified as an independent (0 foreign keys), single (1 foreign key), binary (2 foreign keys) or multiple (more than 2 foreign keys) dependent primary key.

The main objective of this classification is to specify patterns that indicate whether the entities or relations being described in the tables have a dependency on entities/relations described in another table. We achieve this by analyzing the components of the tables and keys.

### 4.2.4 Database views

In addition to relations, views are also described in the database schema. They contain valuable information about further entities described in the database that are of particular interest for the user. Therefore, we are interested in representing this information and include it in the output ontology.

Although database views are similar to tables in the sense that they contain columns and records, they are defined by means of an SQL query and are generated on the fly. Therefore, views are described in a separate branch of the relational-model ontology to reflect their distinctive composition. Database views are instances of the class 'Database view' and are composed by individuals of the class 'Database query'.

A database query (as described in section 2.1) is composed by columns, conditions and joins, which are represented by instances of the classes 'Database attribute', 'Query condition' and 'Query join' respectively. For example, the SQL query that generates the **temporal_outsource_project_filtered** view joins the tables **temporalassignment, project** and **outsourceproject,** and selects the column **project.name** for projects where the **project.cost** column has a value greater than 3000 as follows:

```
SELECT `project`.`name` AS `name`
FROM((`project`
JOIN `temporalassignment` on((`temporalassignment`.`project_id` = `project`.`id`)))
JOIN `outsourceproject` on ((`outsourceproject`.`project_id` = `project`.`id`)))
WHERE (`project`.`cost` > 3000)
```
(18)

The corresponding OWL representation of the SQL query in (18) is an instance of the 'Database query' class which is composed of a column alias, a join and a condition. Axioms representing this query include:

```
Individual: empdb:Temporal_outsource_project_filtered_query
  Types:
   not('has group by attribute' some 'Database attribute'),
   'Database query'
  Facts:
   'has part' empdb:Temporal_outsource_project_filtered_column_alias_1,
   'has part' empdb:Temporal_outsource_project_filtered_condition_1,
```

```
    'has part' Project,
    'has part' empdb:Temporal_outsource_project_filtered_join_1,
    'has part' empdb:Temporal_outsource_project_filtered_join_2,
    not 'has part' dl:all_columns                                          (19)
Individual: empdb:Temporal_outsource_project_filtered_condition_1
   Types:
    'Database query condition'
   Facts:
    'has part'empdb:Project.cost,
    dl:hasOperator ">"^^xsd:string,
    'has value' "3000"^^xsd:double
```

Note in (19) that the assertion not('has group by attribute' some 'Database attribute') indicates that the query does not contain a **group by** function, which is important for preventing the reasoner from classifying this view as an aggregate view (see subsection 4.3.3 for treatment) under OWL's Open World Assumption. In contrast, a 'Database query' that contains aggregation functions makes use of additional properties such as 'has group by attribute' and 'has aggregate function', which is the case for the query in the **managers_by_time** view described as follows:

```
SELECT `manager`.`time_assigned` AS `time_assigned`,
count(0) AS `count(*)`                                                     (20)
FROM `manager`
GROUP BY `manager`.`time_assigned`
```

Axioms representing the aggregation function of the query in (20) are:

```
Individual: empdb:Managers_by_time_query
   Facts:
    'has group by attribute' empdb:manager.time_assigned,
    'has part' empdb:Managers_by_time_column_alias_2,                      (21)
Individual: empdb:Managers_by_time_column_alias_2
   Facts:
    'has aggregate function' "Count"^^xsd:string
```

Once the database components (tables, columns, keys) are classified based on their composition, DBOwlizer uses this classification to map to different OWL constructs using the relational-to-ontology-mapping ontology described in the next subsection.

## 4.3 The Relational to Ontology Mapping Ontology

The combination of the relational-to-ontology-mapping OWL ontology and SWRL rules is used to capture mappings from relational databases to OWL ontologies. In DBOwlizer SWRL rules are instrumental to ground and manipulate more than one variable at a time. The relational-to-ontology-mapping-complex ontology consists of 26 classes, 22 object properties, 1 data property, 2 individuals and corresponds to a $\mathcal{ALCHI}$ description logic. 'Database to Semantic Web mapping' subsumes classes that represent mappings to explicit OWL constructs, such as OWL class, object property, data property, etc. Each one of these classes is automatically populated using their definition (*i.e.* necessary and sufficient conditions).

The definition for each class varies depending on the heuristics being applied, and therefore it is described on the application layer of this ontology (according to the multi-layer design). DBOwlizer's default heuristics are specified in the relational-to-ontology-mapping-dbowlizer ontology. Each heuristic rule is represented as logical axioms in the definition of the corresponding mapping class. Classes in this ontology are populated by individuals representing components of the database schema being mapped, for instance the individuals of the employees-relational-model-individuals ontology. In addition, these individuals need to be classified as instances of the

relational-model-complex ontology classes. Therefore, relational-to-ontology-mapping-dbowlizer imports the relational-to-ontology-mapping-complex, employees-relational-model-individuals and relational-model-complex ontologies which contribute to a total of 81 classes, 39 object properties, 7 data properties, 173 individuals and 25 rules, corresponding to the $\mathcal{ALCHOIQ}(\mathcal{D})$ description logic family. Specific mapping classes to create the output ontology are described using DBOwlizer's default heuristics in the following subsections to illustrate the representation of the heuristics and the heuristics themselves.

The DBOwlizer output ontology is named using the pattern database-mapped-by-methodology, where the word *database* is replaced with the name of the database name being mapped and the word *methodology* is replaced by the name of the set of heuristics being applied. The multi-layer design is also followed in the generation of the output ontology. For example, the output ontology containing the individuals created from the employees database using the default heuristics is: employees-mapped-by-dbowlizer-individuals.owl.

The first OWL constructs created in the DBOwlizer output ontology are the OWL classes. The following subsection describes how classes are created.

### 4.3.1 Mapping to OWL Classes

The class 'Class mapping' specifies the generation of an OWL class in the output ontology for each one of its instances. It is automatically populated by the members of its two subclasses: 'Entity class mapping' and 'Relation class mapping'.

An 'Entity class mapping' will create an OWL class asserted as subclass of the 'informational entity' described in the SIO ontology. Informally, 'Entity Class mapping' will map into a class those tables that represent entities in the database according to the heuristics being applied. In the DBOwlizer's default heuristics, this class is defined as the union of the possible relations whose patterns represent an entity, such as the 'Database unary relationship relation' class (see C1 in Table 6). Notice how the description of this class relies on the inferences previously obtained in the relational-model ontology. For example, the table **manager**, is classified as an instance of the 'Entity class mapping' class because it was previously classified as an instance of the 'Database unary relationship relation' class (refer to Table 4). The table **manager** is be represented in the output ontology as:

```
Class: Manager
SubClassOf:
    'informational entity'                                    (22)
    Employee
```

The subclass relation between the class Manager and Employee is also asserted in (22). This assertion is generated from the membership of the individual empdb:Manager defined in (15) into the 'Subclass mapping' class. In DBOwlizer's default heuristics, the instances of this class are entities completely identified by another entity, *i.e.* tables with a pattern in which the primary key is also a foreign key to another table. These tables are instances of the 'Database unary relationship relation'(see C3 in Table

6). Additionally, the SWRL rule described in (23) infers the relation dl:subClassOfMapping between Manager and Employee.

'Database foreign key'(?fk1) , 'Database primary and foreign key attribute'(?at1) ,
'Database primary key'(?pk2) , 'Database relation'(?r1) , 'Database relation'(?r2) , 'Database single dependent primary key'(?pk1) , 'Subclass mapping'(?r1) , 'has part'(?fk1, ?at1) ,
'has part'(?pk1, ?at1) , 'has part'(?r1, ?at1) , 'has part'(?r2, ?pk2) , references(?fk1, ?pk2) ->
'Unary object property mapping'(?at1) ^ 'sub class of mapping'(?r1, ?r2) (23)

'Relation Class mapping' also creates an OWL class for each one of its instances. Informally, members of this class represent an n-ary relationship with n>2 or a binary relation that contains additional information about this relationship. These relationships cannot be represented with an object property in OWL and thus are represented with an OWL class. In DBOwlizer's default heuristics, such a class is defined as the union of the classes describing tables that represent these kinds of relationships in the database (see C2 in Table 6), for example the 'Database n-ary relationship relation'. The created OWL class is asserted as a subclass of the SIO ontology class 'representational entity'. Given that this class represents a relationship between entities, the required participation of these entities in this relationship must be explicitly stated with a min 1 cardinality constraint. The rationale behind this decision is that a relationship cannot exist without its participants. For example, every individual of the Outsourceworkers class should be related to an individual of the Project class through the property dl:hasOutsourceworkers.project_id. This restriction is represented as follows:

Class: Outsourceworkers
SubClassOf:
'representational entity'
dl:hasOutsourceworkers.project_id min 1 Project (24)

### 4.3.2 Mapping to OWL properties

An individual classified as an instance of 'Object property mapping' is mapped to an OWL object property (relating two entities) in the output ontology. In the context of the relational model, this class represents a binary relation. More specific classes, such as 'Binary Object Property' maps each individual to an object property and asserts an inverse property using the property 'has inverse property mapping'. According to DBOwlizer's default heuristics (see C5 in Table 6), this class is instantiated by individuals representing the columns of a 'Database binary relationship relation' without non-key attributes. This pattern represents binary relations (see Table 3). In the employees database, the columns of the **subproject** table satisfy these restrictions and are therefore mapped into two inverse object properties. One of these object properties is shown in (26).

The DBOwlizer framework creates an object property for each one of the instances of the class 'Unary object property mapping'. The corresponding heuristics, described in C4 in Table 6, classify as instances of this class those individuals representing the table columns that are part of a foreign key with the exception of those classified as a 'Binary object property mapping'. The domain of the object property is the ontology class mapped from the table that contains the foreign key columns, and the range is asserted as the class mapped from the table containing the referenced columns. Domain and range are inferred using the following rule:

```
'Database foreign key'(?ofk) , 'Database primary key'(?pkr) , 'Database relation'(?dr) ,
'Database relation'(?rr) , 'Unary object property mapping'(?op) , 'is part of'(?op, ?dr) ,
'is part of'(?op, ?ofk) , 'is part of'(?pkr, ?rr) , references(?ofk, ?pkr) ->
'has domain mapping'(?op, ?dr) ^ 'has range mapping'(?op, ?rr)
```
(25)

Subclasses of the 'Object property mapping' class are used to specify characteristics of the mapped object properties according to the semantics of the relational model. These characteristics include asserting an object property as functional or transitive (in the logical sense). In the DBOwlizer's default heuristics, instances of 'Unary object property mapping' that reference some primary key are asserted as functional, by means of classifying them as instances of the 'Functional object property mapping' (see C6 in Table 6). Similarly, the 'Inverse functional object property mapping' class includes as instances the individuals representing primary key columns and mapped to object properties (see C7 in Table 6). 'Reflexive object property mapping' describes columns that are part of a foreign key referring to the same table (see R3 in Table 7). Finally, 'Transitive object property mapping' describes a special type of 'Binary object property mapping' whose foreign keys refer to the same table (see R2 in Table 7). For instance, the Subproject.project column is mapped to the following OWL object property:

```
ObjectProperty: dl:hasSubproject.project
  Characteristics:
   Transitive
  Domain: dl:Project
  Range: dl:Project
  InverseOf:
   dl:hasSubproject.subproject
```
(26)

### 4.3.3 Mapping to virtual concepts, properties and collections

A distinctive characteristic of virtual concepts, properties and collections created in the output ontology is that they do not have any instances asserted. On the contrary, instances of these classes will be inferred by a reasoner given their description. This feature is very similar to views in relational databases. The class 'Virtual mapping' subsumes all the classes creating virtual mappings and is populated by the members of its subclasses. The first subclass, 'Virtual class mapping', specifies the generation of an OWL class for each one of its instances. In the DBOwlizer's default heuristics (see C11 in Table 6), all views are inferred as instances of this class, with the exception of those described by queries with aggregation functions or with tables mapped to properties. In addition to the generation of an OWL class in the output ontology, if a join is part of the view query, additional classes are created to represent the intersection of classes defined in the join. For example, the view **high_salary_employee,** which retrieves employees whose salary is greater than or equal to 80,000, is classified as a member of the 'Virtual class mapping' and mapped to the class:

```
Class: dl:High_salary_employee_salary
EquivalentTo:
    (Employee
     and (dl:hasEmployee.salary some xsd:double[>= 80,000]))
     and (dl:hasEmployee.salary some rdfs:Literal)
```
(27)

The class 'Virtual property mapping' generates an object property for each one of its members. In DBOwlizer's default heuristics, this class is populated by views that contain a table mapped to an object property (see R18 in Table 7). Such views are mapped to a sub object property of such an object property with a more restricted

domain. The rationale behind this is that they represent a more specific relationship between two entities, *e.g.* worksfor_dept2 is mapped to the object property defined in (28), whose domain is restricted to instances of the class Employee related to the individual department_2 via the hasWorksfor.department_id property.

```
ObjectProperty: dl:Worksfor_dept2
  SubPropertyOf:
    dl:hasWorksfor.department_id
  Domain: Worksfor_dept2_domain
```
(28)

Finally, the 'Virtual collection mapping' specifies the generation of classes and individuals representing a collection of groups for each one of its instances. In DBOwlizer's default heuristics, views with an aggregation function ('Database aggregation view') are classified as members of this class (see C12 in Table 6). For example, the view **managers_by_time** of the employees database is defined by the query in (20). This query creates groups of records of **manager** given their value in the **manager.time_assigned** column and returns the count for each group. This view is mapped to the individual empdb:Managers_by_time and related to the group of rows with **"FULL-TIME"** value represented by the individual empdb:Managers_by_time_group_by_full-time. Both the numeric result of the aggregation function, which in this case is 2, and the members of the group are also represented by using the 'has count' data property and the defined class dl:Managers_by_time_group_by_full-time_class respectively. The number of instances of this class corresponds to the value of the count function asserted. Axioms of the mapped classes and individuals include:

```
Individual: empdb:Managers_by_time
  Types:
    'Manager Collection'
  Facts:
    'has part'empdb:Managers_by_time_group_by_full-time,
    'has part'empdb:Managers_by_time_group_by_part-time
Individual: empdb:Managers_by_time_group_by_full-time
  Types:                                                                    (29)
    'Manager Group',
    'has member' some dl:Managers_by_time_group_by_full-time_class
  Facts:
    dl:hasCount 2
Class: dl:Managers_by_time_group_by_full-time_class
  EquivalentTo:
    Manager
    and (dl:hasManager.time_assigned value "FULL-TIME")
```

Table 6 includes the formal description in Manchester OWL syntax of the mapping classes according to DBOwlizer's default heuristics with a unique identifier for each class.

Similarly, Table 7 contains a list of the SWRL rules used by DBOwlizer's default heuristics in the human readable syntax provided by Protégé with a description of the mapping defined by each rule. Each rule has a unique identifier.

For all the axioms in this ontology, please refer to the relational-to-ontology-mapping-dbowlizer ontology.

Table 6. Class definition of the mapping classes in the ontology relational-to-ontology-mapping-dbowlizer in Manchester OWL syntax. Each definition encodes the default heuristics to automatically generate OWL ontologies from relational databases in DBOwlizer.

| Rule id | Mapping class description |
|---|---|
| C1 | **'Entity class mapping'** EquivalentTo:<br>'Database partially unary relationship relation'<br>or 'Database entity relation' or ('Database unary relationship relation'<br>and ('has part' some ('Database primary key'<br>and ('has part' max 1 'Database attribute')))) |
| C2 | **'Relation class mapping'** EquivalentTo:<br>'Database n-ary relationship relation'<br>or 'Database partially binary relationship relation'<br>or 'Database partially n-ary relationship relation'<br>or ('Database binary relationship relation'<br>and ('has part' some 'Database non-key attribute')) |
| C3 | **'Subclass mapping'** EquivalentTo:<br>'Database unary relationship relation'<br>and ('has part' some ('Database primary key'<br>and ('has attribute' max 1 'Database attribute'))) |
| C4 | **'Unary object property mapping'** EquivalentTo:<br>'Database non-primary and foreign key attribute'<br>or ('Database primary and foreign key attribute'<br>and ('is part of' some<br>('Database multiple dependent primary key'<br>or 'Database partially binary dependent primary key'<br>or 'Database partially multiple dependent primary key'<br>or 'Database partially single dependent primary key'<br>or ('Database binary dependent primary key'<br>and ('is part of' some ('Database relation'<br>and ('has part' some 'Database non-key attribute')))))) |
| C5 | **'Binary object property mapping'** EquivalentTo:<br>'Database primary and foreign key attribute'<br>and ('is part of' some ('Database binary relationship relation'<br>and (not ('has part' some 'Database non-key attribute')))) |
| C6 | **'Functional object property mapping'** EquivalentTo:<br>'Unary object property mapping'<br>and ('is part of' some (references some 'Database primary key')) |
| C7 | **'Inverse functional object property mapping'** EquivalentTo:<br>'Object property mapping' and ('is part of' some 'Database primary key') |
| C8 | **'Data property mapping'** EquivalentTo:<br>'Database non foreign key attribute' |
| C9 | **'Functional data property mapping'** EquivalentTo:<br>'Database not-null attribute' and 'Data property mapping' |
| C10 | **'Data property mapping with ennumeration range'** Equivalent To:<br>'Data property mapping' and (dl:hasValue some Literal) |
| C11 | **'Virtual class mapping'** EquivalentTo:<br>'Database equivalent view' or 'Database projection and selection view'<br>or 'Database projection view' or ('Database selection view' |

| | |
|---|---|
| | and (not ('Virtual property mapping'))) |
| C12 | **'Virtual collection mapping'** EquivalentTo:<br>'Database aggregation view' and ('has part' some<br>('Database query' and ('has part' some<br>('Database attribute alias' and ('has part' some ({dl:all_columns}))))))) |

Table 7. List of the SWRL rules included in DBOwlizer's default heuristics using the human readable syntax provided by Protégé.

| Rule id | Description | Rule |
|---|---|---|
| R1 | Defines the domain and range of a unary object property. | 'Database foreign key'(?ofk) , 'Database primary key'(?pkr) , 'Database relation'(?dr) , 'Database relation'(?rr) , 'Unary object property mapping'(?op) , 'is part of'(?op, ?dr) , 'is part of'(?op, ?ofk) , 'is part of'(?pkr, ?rr) , references(?ofk, ?pkr) -><br>'has domain mapping'(?op, ?dr) ∧ 'has range mapping'(?op, ?rr) |
| R2 | Defines a transitive object property. | 'Database foreign key'(?fk1) , 'Database foreign key'(?fk2) , 'Database primary and foreign key attribute'(?at1) , 'Database primary and foreign key attribute'(?at2) , 'Database relation'(?r) , 'Database single dependent primary key'(?pk) , 'has part'(?fk1, ?at1) , 'has part'(?fk2, ?at2) , 'has part'(?pk, ?at1) , 'has part'(?pk, ?at2) , 'has part'(?r, ?at1) , 'has part'(?r, ?at2) , differentFrom(?at1, ?at2) , differentFrom(?fk1, ?fk2) -><br>'Binary object property mapping'(?at1) ∧<br>'Transitive object property mapping'(?at1) |
| R3 | Defines a reflexive object property. | 'Database foreign key'(?fk1) , 'Database foreign key attribute'(?at1) , 'Database primary key'(?pk2) , 'Database relation'(?r) , 'has part'(?fk1, ?at1) , 'has part'(?r, ?fk1) , 'has part'(?r, ?pk2) , references(?fk1, ?pk2) -><br>'Reflexive object property mapping'(?at1) |
| R4 | Defines the subclass mapping between two relations. | 'Database foreign key'(?fk1) , 'Database primary and foreign key attribute'(?at1) , 'Database primary key'(?pk2) , 'Database relation'(?r1) , 'Database relation'(?r2) , 'Database single dependent primary key'(?pk1) , 'Subclass mapping'(?r1) , 'has part'(?fk1, ?at1) , 'has part'(?pk1, ?at1) , 'has part'(?r1, ?at1) , 'has part'(?r2, ?pk2) , references(?fk1, ?pk2) -><br>'Unary object property mapping'(?at1) ∧ 'sub class of mapping'(?r1, ?r2) |
| R5 | Defines an existential restriction over an n-ary relationship. | 'Database foreign key'(?opk) , 'Database n-ary relationship relation'(?opr) , 'Unary object property mapping'(?op) , 'is part of'(?op, ?opk) , 'is part of'(?opk, ?opr) -> 'exists some restriction mapping'(?opr, ?op) |
| R6 | Defines the domain and range of a data property. | 'Database relation'(?dr) , 'Data property mapping'(?dp) , dl:hasAttributeDomain(?dp, ?dt) , 'is part of'(?dp, ?dr) -> 'has domain mapping'(?dp, ?dr) ∧ 'has range mapping'(?dp, ?dt) |
| R7 | Defines min 1 cardinality for an attribute with not null restrictions. | 'Database not-null attribute'(?op) , 'Database relation'(?dr) , 'Object property mapping'(?op) , 'has part'(?dr, ?op) -><br>'min 1 restriction'(?dr, ?op) |
| R8 | Defines the domain, range and inverse property of a binary object property. | 'Binary object property mapping'(?op1) , 'Binary object property mapping'(?op2) , 'Database foreign key'(?ofk1) , 'Database foreign key'(?ofk2) , 'Database primary key'(?pkr1) , 'Database primary key'(?pkr2) , 'Database relation'(?br) , 'Database relation'(?rr1) , 'Database relation'(?rr2) , 'is part of'(?op1, ?br) , 'is part of'(?op1, ?ofk1) , 'is part |

| | | |
|---|---|---|
| | | of'(?op2, ?br) , 'is part of'(?op2, ?ofk2) , 'is part of'(?pkr1, ?rr1) , 'is part of'(?pkr2, ?rr2) , references(?ofk1, ?pkr1) , references(?ofk2, ?pkr2) , differentFrom(?op1, ?op2) -> 'has domain mapping'(?op1, ?rr2) ∧ 'has inverse property mapping'(?op1, ?op2) ∧ 'has range mapping'(?op1, ?rr1) |
| R9 | Defines a virtual class from a view with selection. | 'Class mapping'(?r) , 'Database query'(?q) , 'Database relation'(?r) , 'Database selection view'(?v) , 'has part'(?q, ?r) , 'has part'(?v, ?q) -> 'Virtual class mapping'(?v) |
| R10 | Defines an equivalent class from an equivalent view. | 'Database equivalent view'(?v) , 'Database query'(?q) , 'Database relation'(?r) , 'Virtual class mapping'(?v) , 'has part'(?q, ?r) , 'has part'(?v, ?q) -> 'equivalent class mapping'(?v, ?r) |
| R11 | Defines a subclass used as property domain from a view with join and selection. | 'Database attribute'(?at) , 'Database attribute'(?at1) , 'Database join and selection view'(?v) , 'Database query'(?q) , 'Database query condition'(?c) , 'Database query join'(?j) , 'Database relation'(?r) , 'has domain mapping'(?j, ?r) , 'has part'(?c, ?at1) , 'has part'(?q, ?c) , 'has part'(?q, ?j) , 'has part'(?r, ?at) , 'has part'(?r, ?at1) , 'has part'(?v, ?q) -> 'is sub class of domain mapping'(?c, ?j) ∧ 'sub class of mapping'(?c, ?r) |
| R12 | Defines a subclass used as property range from a view with join and projection. | 'Database attribute'(?at) , 'Database attribute alias'(?al) , 'Database join projection view'(?v) , 'Database query'(?q) , 'Database query join'(?j) , 'Database relation'(?r) , 'has part'(?al, ?at) , 'has part'(?q, ?al) , 'has part'(?q, ?j) , 'has part'(?r, ?at) , 'has part'(?v, ?q) , 'has range mapping'(?j, ?r) -> 'is sub class of range mapping'(?al, ?j) ∧ 'property restricted concept mapping'(?al, ?r) |
| R13 | Defines a subclass used as property range from a view with join, projection and selection. | 'Database attribute'(?at) , 'Database attribute'(?at1) , 'Database join selection projection view'(?v) , 'Database query'(?q) , 'Database query condition'(?c) , 'Database query join'(?j) , 'Database relation'(?r) , 'has part'(?c, ?at1) , 'has part'(?q, ?c) , 'has part'(?q, ?j) , 'has part'(?r, ?at) , 'has part'(?r, ?at1) , 'has part'(?v, ?q) , 'has range mapping'(?j, ?r) -> 'is sub class of range mapping'(?c, ?j) ∧ 'sub class of mapping'(?c, ?r) |
| R14 | Defines a subclass from a view with projection and selection. | 'Database projection and selection view'(?v) , 'Database query'(?cm) , 'Database relation'(?r) , 'Virtual class mapping'(?v) , 'has part'(?cm, ?r) , 'has part'(?v, ?cm) -> 'property restricted concept mapping'(?v, ?r) ∧ 'sub class of mapping'(?v, ?r) |
| R15 | Defines a property, domain and range from a view with a mapping to data property. | 'Database query join'(?j) , 'Database relation'(?r1) , 'Database relation'(?r2) , 'Data property mapping'(?at2) , 'has part'(?j, ?at1) , 'has part'(?j, ?at2) , 'has part'(?j, ?r1) , 'has part'(?r1, ?at1) , 'has part'(?r2, ?at2) , differentFrom(?at1, ?at2) , differentFrom(?r1, ?r2) -> 'has domain mapping'(?j, ?r1) ∧ 'has property mapping'(?j, ?at1) ∧ 'has range mapping'(?j, ?r2) |
| R16 | Defines a subclass used as property range from a view with join and selection. | 'Database attribute'(?at) , 'Database attribute'(?at1) , 'Database join and selection view'(?v) , 'Database query'(?q) , 'Database query condition'(?c) , 'Database query join'(?j) , 'Database relation'(?r) , 'has part'(?c, ?at1) , 'has part'(?q, ?c) , 'has part'(?q, ?j) , 'has part'(?r, ?at) , 'has part'(?r, ?at1) , 'has part'(?v, ?q) , 'has range mapping'(?j, ?r) -> 'is sub class of range mapping'(?c, ?j) ∧ 'sub class of mapping'(?c, ?r) |
| R17 | Defines a subclass relation between a view and the table from which is generated. | 'Database query'(?cm) , 'Database relation'(?r) , 'Database selection view'(?v) , 'Virtual class mapping'(?v) , 'has part'(?cm, ?r) , 'has part'(?v, ?cm) -> 'sub class of mapping'(?v, ?r) |

| | | |
|---|---|---|
| R18 | Defines a property from a view with selection and mapping to binary object properties. | 'Binary object property mapping'(?at) , 'Database attribute'(?at) , 'Database query'(?q) , 'Database query condition'(?c) , 'Database selection view'(?v) , 'has part'(?c, ?at) , 'has part'(?q, ?c) , 'has part'(?v, ?q) -> 'Virtual property mapping'(?v) |
| R19 | Defines a subclass used as property domain from a view with join and projection. | 'Database attribute'(?at) , 'Database attribute alias'(?al) , 'Database join projection view'(?v) , 'Database query'(?q) , 'Database query join'(?j) , 'Database relation'(?r) , 'has domain mapping'(?j, ?r) , 'has part'(?al, ?at) , 'has part'(?q, ?al) , 'has part'(?q, ?j) , 'has part'(?r, ?at) , 'has part'(?v, ?q) -> 'is sub class of domain mapping'(?al, ?j) ∧ 'property restricted concept mapping'(?al, ?r) |
| R20 | Defines a property, domain and range from a query join. | 'Database primary key attribute'(?at2) , 'Database query join'(?j) , 'Database relation'(?r1) , 'Database relation'(?r2) , 'has part'(?j, ?at1) , 'has part'(?j, ?at2) , 'has part'(?j, ?r1) , 'has part'(?r1, ?at1) , 'has part'(?r2, ?at2) , differentFrom(?at1, ?at2) -> 'has domain mapping'(?j, ?r1) ∧ 'has property mapping'(?j, ?at1) ∧ 'has range mapping'(?j, ?r2) |
| R21 | Defines a subclass used as property domain from a view with join, projection, selection and alias. | 'Database attribute'(?at) , 'Database attribute alias'(?al) , 'Database join selection projection view'(?v) , 'Database query'(?q) , 'Database query join'(?j) , 'Database relation'(?r) , 'has domain mapping'(?j, ?r) , 'has part'(?al, ?at) , 'has part'(?q, ?al) , 'has part'(?q, ?j) , 'has part'(?r, ?at) , 'has part'(?v, ?q) -> 'is sub class of domain mapping'(?al, ?j) ∧ 'property restricted concept mapping'(?al, ?r) |
| R22 | Defines a subclass used as property range from a view with join, projection, selection and alias. | 'Database attribute'(?at) , 'Database attribute alias'(?al) , 'Database join selection projection view'(?v) , 'Database query'(?q) , 'Database query join'(?j) , 'Database relation'(?r) , 'has part'(?al, ?at) , 'has part'(?q, ?al) , 'has part'(?q, ?j) , 'has part'(?r, ?at) , 'has part'(?v, ?q) , 'has range mapping'(?j, ?r) -> 'is sub class of range mapping'(?al, ?j) ∧ 'property restricted concept mapping'(?al, ?r) |
| R23 | Defines a subclass from a view with projection. | 'Databas e projection view'(?v) , 'Database query'(?cm) , 'Database relation'(?r) , 'Virtual class mapping'(?v) , 'has part'(?cm, ?r) , 'has part'(?v, ?cm) -> 'property restricted concept mapping'(?v, ?r) |
| R24 | Defines a subclass used as a property domain from a view with join, projection and selection. | 'Database attribute'(?at) , 'Database attribute'(?at1) , 'Database join selection projection view'(?v) , 'Database query'(?q) , 'Database query condition'(?c) , 'Database query join'(?j) , 'Database relation'(?r) , 'has domain mapping'(?j, ?r) , 'has part'(?c, ?at1) , 'has part'(?q, ?c) , 'has part'(?q, ?j) , 'has part'(?r, ?at) , 'has part'(?r, ?at1) , 'has part'(?v, ?q) -> 'is sub class of domain mapping'(?c, ?j) ∧ 'sub class of mapping'(?c, ?r) |
| R25 | Defines a property, domain and range from a view with a mapping to object property. | 'Database query join'(?j) , 'Database relation'(?r1) , 'Database relation'(?r2) , 'Object property mapping'(?at2) , 'has part'(?j, ?at1) , 'has part'(?j, ?at2) , 'has part'(?j, ?r1) , 'has part'(?r1, ?at1) , 'has range mapping'(?at2, ?r2) , differentFrom(?at1, ?at2) -> 'has domain mapping'(?j, ?r1) ∧ 'has property mapping'(?j, ?at1) ∧ 'has range mapping'(?j, ?r2) |

### 4.3.4 Mapping Individuals

DBOwlizer maps table rows into specific entities with attributes and relations between them. This is achieved through the generation of individuals, data properties (and their corresponding values) and object properties in the output ontology.

Each table row is mapped to an individual whose identifier contains the primary key value identifying the row. This mapping is more appropriate when business keys (such as social insurance numbers) are used to represent entities. For example, the individual created from the row of the table **manager** whose primary key value is "**1**" has the identifier: empdb:manager_1. Individuals are asserted as instances of the OWL class generated from the table they belong to. For instance, empdb:manager_1 is asserted as an instance of the class dl:Manager in (30).

When a column is mapped to a data property, such a data property links the value in the cell of the row corresponding to that column. For example, the column **manager.entry_date** is mapped to the data property dl:hasManager.entry_date and relates the individual empdb:manager_1 to its corresponding value and datatype "2009-01-01"^^xsd:date in (30).

Similarly, when a column is mapped to an object property, individuals are linked through such an object property considering foreign keys. For example the column **manager.project** is mapped to the object property dl:hasManager.project and links the individual empdb:manager_1 with the individual empdb:project_1 in (30).

```
Individual: empdb:manager_1
  Types:                                                    (30)
    dl:Manager
  Facts:
   dl:hasManager.id empdb:employee_1,
   dl:hasManager.project empdb:project_1,
   dl:hasManager.entry_date "2009-01-01"^^xsd:date,
   dl:hasManager.time_assigned "FULL-TIME"^^xsd:string
```

Note that no individual is asserted as instance of the classes created from the class 'Virtual mapping', as explained in subsection 4.3.3.

## 4.4 Discussion

In this Chapter the representation, automatic generation and execution of mappings from a database to an OWL ontology in DBOwlizer have been described and illustrated with the use of a test case. DBOwlizer uses ontologies to represent and classify database schema components. Distinctive characteristics of these features are discussed in the following subsections.

### 4.4.1 Representing mappings using OWL ontologies and SWRL rules

As described in section 4.3, database to OWL mappings in DBOwlizer are represented by using OWL ontologies themselves. Some of the advantages of this representation are: i) it bypasses the need for another representation language by using OWL and SWRL, widely known by the semantic web community and for which human-friendly representations such as Manchester OWL syntax are available, ii) it facilitates modifications/updates to mapping heuristic rules, since modifications to heuristics rules require only modification to class definitions (which can be done in an OWL ontology

editor such as Protégé) and not modification of source code as related approaches do, iii) it facilitates the reuse or integration of mapping heuristic rules, since rules are represented in OWL and their restrictions can easily be imported by other ontologies, iv) it facilitates the comparison of different mapping heuristics due to the multi-layer ontology design; this design allows the reuse of the vocabulary and previous inferences in the primitive and complex layers and a database test case, v) it separates domain knowledge from model knowledge (*e.g.* the OWL classes `table` and `column` are not siblings of the class `employee` in the output ontology) and vi) it can be easily extended to accommodate additional data models, by replacing the `relational-model` ontology with an ontology describing the new data model and providing an implementation to access the corresponding data.

One of the challenges of DBOwlizer's mapping representation is to reconcile the Closed World assumption that holds in the relational model with the Open World Assumption in OWL. For instance, in order to obtain an aggregate value from a group of rows in the database, it becomes necessary to assume that the rows found in the table are the only rows in the database, in order to calculate the aggregated value. Hence, some of the class definitions, require the use of negated axioms or cardinality constraints with the exactly restriction. For example, the representation of the query in (19) includes the statement `not('has group by attribute' some 'Database attribute')` to prevent the reasoner from inferring that this query has an aggregation function. DBOwlizer also instantiates specific classes that need closure axioms, *e.g.* the primary key of the **employee** table (Figure 9), `empdb:employee.primary_key`, is asserted as a 'Database

independent primary key' because it contains *only* attributes that do not refer to other relations (*i.e. not* part of any foreign key). The use of reasoners such as TrOWL which allows closed world reasoning will eliminate the need for these assertions with the use of annotations.

Similarly, OWL does not hold the Unique Name Assumption. This is also a necessary feature for aggregation functions in a database. In order to obtain this behaviour we must explicitly assert that the individuals in the ontology are different one from another using the OWL construct `DifferentIndividuals`.

Note that the `relational-to-ontology-mapping` ontology can be seen as a "meta-ontology", since it describes the elements of the relational model and how to map to OWL components. However, to prevent the generation of OWL Full ontologies, DBOwlizer never refers to the same entity as an individual and as a class in the same ontology. For example, the table **employee** is considered as an instance for mapping purposes and it is also represented as an OWL class in the output ontology, which is never merged with the mapping ontologies. An alternative design for meta-modelling would be punning (refer to subsection 2.2.2).

The DBOwlizer approach takes a unique perspective on representing SQL queries containing aggregation functions. The challenge of modelling aggregation functions is that a group of rows is aggregated into a single value produced by such a function. Asserting directly the relation of this value to the OWL class representing the group through a data property would require the ontology to use OWL Full, since the OWL class would be treated as an individual, preventing DBOwlizer from using reasoning

services. Instead, the DBOwlizer design involves the explicit description of groups and collections. Each group is attributed with the aggregated value and is related to the class that defines the characteristics of the group. This group class is dynamically populated, just as the database views. Therefore, DBOwlizer not only explicitly provides the aggregation value for the group, but also the members of such a group.

### 4.4.2 Generation and execution of mappings in DBOwlizer

To automatically generate relational databases to OWL mappings, DBOwlizer's default heuristics have two main assumptions. The first assumption is that the database to be mapped is normalized in at least Third Normal Form and designed following good practices (Elmasri and Navathe 2003). A database whose tables are in Third Normal form guarantees that the information contained in a table in fact belongs to the entity or relationship represented by the table. If such good practices were not followed, flaws will still exist in the output ontology. For example, in a non Third Normal Form version of the employees database, if an attribute related to a department such as the location is included in the employee table, the output ontology will assign the location to the class representing employees instead of the class representing department. However, a non-normalized database can be automatically normalized using a tool such a JMathNorm (Yazici and Karakaya 2007). The second assumption is that the primary keys and foreign keys are explicitly declared, such as in the employees database test case. The composition of a table in terms of its keys in addition to its tables is crucial in the mapping classification as described in section 4.3.

A distinctive feature in the DBOwlizer mapping representation is the representation of the heuristic rules as necessary and sufficient conditions of the mapping classes. Although a default mapping is provided, these rules can be manually modified at any point in the process and the output ontology will reflect these changes. A disadvantage of this representation is that in order to automatically populate each mapping class, a DL reasoner is required. Therefore the scalability of this process is highly dependent on the scalability of reasoners' classification tasks. For example, using default heuristics and the employees database, HermiT generates the output ontology five times faster than Pellet.

When creating the output ontology, OWL classes created from an 'Entity class mapping' are asserted as a subclass of SIO's 'informational entity'. In SIO, an 'informational entity' is an entity that is an abstraction of, representation for, description of or information about some entity (whether a type or instance), and only has informational entities as its parts, which is consistent with the representation of entities in the relational model. Moreover, database records contain information about an entity, and not the entity itself. For example, the Employee class in the output ontology contains information about the employee named "Joe", but it is not Joe himself. Similarly, classes created from a 'Relation Class Mapping' are asserted as a subclass of the SIO's class 'representational entity', which in SIO is defined as an entity that in some way represents another entity. In our context, it may represent an entity and its parts, or a relationship between entities. Further refinement using

ontology mapping with already existing ontologies may help to refine add more semantics to this class.

### 4.4.3 Limitations of the mapping generation

It should be noted that if the targeted relational database does not contain patterns describing class hierarchies or more complex concepts, the output ontology will reflect the simple expressivity of the database.

When the same table pattern is used two represent more than one type of relationship, the DBOwlizer default mapping heuristics fails to discriminate between these relationships. For example, the pattern to recognize a 'Subclass mapping' defined in subsection 4.3.1 can also be used to represent table partitions and therefore parts of the entity or process being defined in a table. DBOwlizer's default heuristics are not able to recognize these alternative meanings and map this pattern into a subclass relationship.

As described in this chapter, DBOwlizer's default heuristics analyze a database schema based on the SQL constructs used for its definition. These constructs are known as data definition and data manipulation statements (Groff, Weinberg et al. 2010). Specific constructs covered by DBOwlizer are listed in Table 8, which contains a summary of SQL statement constructs and datatypes from (Groff, Weinberg et al. 2010).

SQL constructs for data access and transaction control are not covered by DBOwlizer's default heuristics given that they are used to operate the database at the implementation level and do not contain semantics about the data being stored.

Similarly, standard built functions to manipulate data are not covered by DBOwlizer's default heuristics.

Table 8. SQL constructs covered and not covered by DBOwlizer's default heuristics.

| Construct type | Covered in DBOwlizer | Not Covered in DBOwlizer |
|---|---|---|
| Data Manipulation Statements and Functions | SELECT, FROM, WHERE, CONDITION OPERATORS (=, >, <, <=, >=, <>), EQUI-JOIN (=), * (ALL COLUMNS), SELF-JOINS, TABLE ALIAS, SUM, AVG, MIN, MAX, COUNT , GROUP BY. | INSERT, UPDATE, MERGE, DELETE, DISTINCT, BETWEEN (ranges), LIKE, AND, OR, TRUE, FALSE, UNKNOWN, ORDER BY, ASC, DESC, UNION JOIN, OUTER JOIN, CROSS JOIN, HAVING, SUBQUERIES (NESTED, CORRELATED), EXISTS, ANY, ALL, NULLIF, CASE (WHEN, THEN), BIT_LENGTH, CAST, CHAR_LENGTH, CONVERT, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, EXTRACT, LOWER, OCTET_LENGTH, POSITION, SUBSTRING, TRANSLATE , TRIM. |
| Data Definition Statements | CREATE TABLE, CREATE VIEW, CREATE SCHEMA. | DROP TABLE, ALTER TABLE, DROP VIEW, CREATE INDEX, DROP INDEX, DROP SCHEMA, CREATE DOMAIN, ALTER DOMAIN, DROP DOMAIN. |
| Data Types | NULL, NOT NULL, CHARACTER (CHAR), CHARACTER VARYING (VARCHAR), INTEGER (INT), SMALLINT, DECIMAL, FLOAT, REAL, DOUBLE PRECISION (DOUBLE), DATE, TIME. | BIT, NUMERIC, TIME WITH TIME ZONE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, INTERVAL, XML. |
| Access and Transaction Control | | GRANT, REVOKE, CREATE ROLE, GRANT ROLE, DROP ROLE, COMMIT, ROLLBACK, SET TRANSACTION, START TRANSACTION, SAVE POINT. |

Although some complex SQL constructs do not have a corresponding DBOwlizer heuristic rule, DBOwlizer can be used as building blocks to define such complex constructs. For instance, a nested query can be mapped into a set of OWL classes related by a transitive property. Each subclass in this set would represent each one of the sub-queries. This design is similar to the mapping of joins described in R11 and R16 in Table 7.

**The following chapter will describe the DBOwlizer prototype implemented as a proof of concept of the heuristics described in this chapter.**

# Chapter 5 Implementation

To demonstrate the distinctive features of the DBOwlizer framework a prototype was implemented. This prototype allows a user to: i) map a database schema to an OWL representation, ii) classify the database components according to the `relational-model` ontology and `relational-to-ontology-mapping` ontology using a DL reasoner and iii)

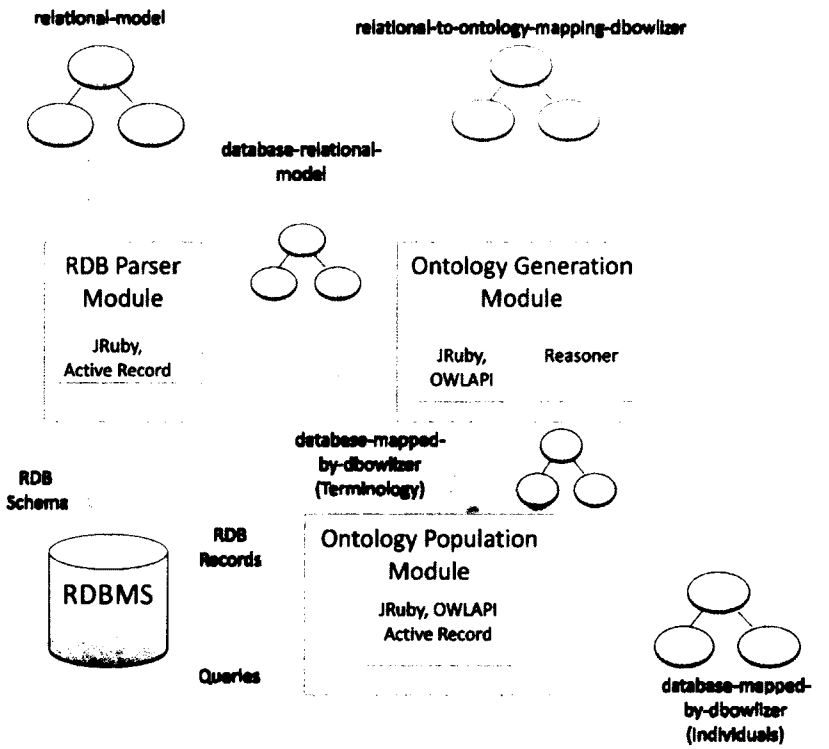



Figure 10. The DBOwlizer prototype contains the Relational Database (RDB) Parser Module, that parses a database schema and translates it to OWL. The Ontology Generation Module creates the terminology of the output ontology while the Ontology Population Module populates it by querying the RDBMS. Ontologies representing the relational database are denoted by *blue trees*, *green trees* represent heuristics and *red trees* denote the output ontology.

execute relational database to OWL mappings to generate an output OWL ontology with the information represented in the database.

The DBOwlizer prototype was developed using JRuby (Edelson and Liu 2008), a Java implementation of the Ruby programming language that allows interoperability with Java code and libraries. Figure 10 shows the modules of this prototype. The implementation contains a total of 29 Ruby classes and 2 configuration files that are grouped in 3 modules, each of which are described in the following section.

## 5.1 DBOwlizer Implementation Modules

The three modules composing the DBOwlizer prototype are the Relational Database Parser, Ontology Generation and Ontology Population modules. The Relational Database Parser Module communicates with the Relational Database Management System (RDBMS) to obtain the description of the database schema. This description is then mapped to its corresponding OWL ontology such as the `employees-relational-model-individuals` ontology. This ontology, along with the mapping heuristics, such as the `relational-to-ontology-mapping-dbowlizer` ontology, is used by the Ontology Generation module to generate the terminology of the output ontology, *e.g.* `employees-mapped-by-dbowlizer-complex`. The output terminology is used to populate the ontology in the Ontology Population module, which interacts with the RDBMS to pose queries and create an ontology containing individuals, such as the `employees-mapped-by-dbowlizer-individuals`. Further details about these modules follow.

### 5.1.1 Relational Database Parser module

The Relational Database (RDB) Parser module communicates with the RDBMS using a JDBC connection. At the time of writing this thesis, it parses database schema scripts from MySQL using the Active Record Ruby library (Bächle and Kirchberg 2007). This library implements the object-relational mapping (ORM) pattern (Fowler 2003), which wraps the database access in an object. Active Record is normally used to create a persistent model to store information about objects in a database. In the DBOwlizer implementation we used it in the opposite way, to pose queries to an already existing database. Therefore, the first step in the process is to dynamically specify the definitions of each table based on the database schema and store it in a file named persister.rb.

The input to this module is the database schema defined by means of SQL CREATE statements, such as the one described in (12). These statements are parsed to generate the database-relational-model-individuals ontology in which database components such as tables and views are represented as instances of classes of the relational-model ontology. For example, the employees database schema is used to generate the employees-relational-model-individuals ontology which contains, for example, the individual empdb:employee as an instance of the class 'Database relation'. This individual represents the table **employee**.

### 5.1.2 Ontology Generation Module

The Ontology Generation Module generates the terminology of the mapped output ontology. To create such a terminology it requires the target database schema

(represented in OWL) and the heuristics that will specify the mappings. Therefore, the input to this module is composed by: i) the database-relational-model-individuals ontology and ii) the ontology with the mapping heuristics, such as the default one described in section 4.3 (i.e. relational-to-ontology-mapping-dbowlizer ontology) or one from related work presented in Chapter 6 or a user customized.

The implementation of this module is based on the OWL-API version 3 (Bechhofer, Noppens et al. 2007) and Pellet to load and classify the ontologies, populate each one of the mapping classes and execute the mappings by means of creating the required terminology in the output ontology database-mapped-by-methodology according to the description in section 4.3. For example, when using DBOwlizer's default heuristics with the employees database, this module creates the ontology employees-mapped-by-dbowlizer-primitive (primitive layer) and the corresponding complex layer if applicable. In DBOwlizer, an OWL class is created for each one of the instances of the class 'Entity class mapping'. The use of the OWL-API facilitates interoperability with OWL applications and programs such as ontology editors and DL reasoners.

Reasoning-specific tasks are the most expensive in our mapping process given the expressivity of the ontologies being manipulated. The ontology relational-to-ontology-mapping-dbowlizer is the most expressive in DBOwlizer and belongs to the Description Logic family $\mathcal{ALCHOIQ}(\mathcal{D})$. For this family, the reasoning problems (*i.e.* ontology consistency, concept satisfiability, concept subsumption and instance checking) have a complexity of NEXPTIME-complete (Schaerf 1994; Cuenca Grau 2007; Motik 2007). Currently, the classification and realization processes for the employee

database using HermiT (version 1.3.4) is 46.225 seconds using a PC Intel Core i7 with 2.67GHz, 6 GB in RAM and Windows Vista 64-bit as operating system. This performance is reasonable given the expressivity of the ontology obtained, and users are advised to keep a balance between expressivity of the output ontology and performance of the mapping process. We intend to pursue more scalable and optimized reasoners although this problem is out of the scope of this thesis.

### 5.1.3 Ontology Population Module

The Ontology Population Module populates the ontology created by the Ontology Generation module. By posing queries to the RDBMS using Active Record, it creates the corresponding individuals and their relations in an independent ontology whose name uses the same pattern as the terminology. For instance, the ontology created from the employees database using the default heuristics is: `employees-mapped-by-dbowlizer-individuals` and contains, among others, the individuals representing the groups and collection of managers described in (29).

Note that the output ontology reflects a *snapshot* of the database at the time of mapping. If the database changes, the output ontology should be updated to reflect this.

The URIs of DBOwlizer ontologies use the prefix `http://semanticscience.org/ontology`. Classes and individuals also have the same prefix with the variation of the word "resource" instead of "ontology", *e.g.* `http://semanticscience.org/resource/dl:DBRelation`. All DBOwlizer classes are also

annotated with a label, *e.g.* 'Database relation', which is used instead of the full URI in this document for readability purposes. Individuals in the output ontology use the same default prefix followed by the name of the database. The naming of resources in this prototype is limited in terms of string manipulation using functions or patterns. Even though these patterns can be represented using SWRL built-ins, this feature was not available at the time of development in Pellet 1.5, the DL reasoner used in the DBOwlizer prototype. It will be added to the framework when available. We followed a naming convention similar to the one described in (Villanueva-Rosales, Osbahr et al. 2007) for data mashup in life sciences, where legacy information is included in the URI of a resource, such as its source and provider (*e.g.* database name and methodology for mapping). Note however, that in the DBOwlizer framework the name of database components and ontology resources is meaningless. The mapping heuristics and ontology classification is based on the composition of the entities and not on their names.

## 5.2 Configuration

The DBOwlizer prototype has predefined prefixes and naming conventions but it can also be configurable to a great extent in the database.yml and ontology.yml (text) files.

In the database.yml file, the database access details are specified, such as database name, username and password. In the ontology.yml file, elements for the generation of resources are specified: directories, prefixes, variables and classes. The directory to create the output ontology, and the URI or directory of the ontology containing the

mapping heuristics can also be modified. Similarly, class names for the relational-model and relational-to-ontology-mapping ontologies such as 'Database relation', 'Virtual class mapping' or SIO's 'informational entity' can be changed to accommodate modifications to the DBOwlizer framework ontologies. If a change in the DBOwlizer mapping ontologies is required, by modifying this file, the parsing and generation of output ontologies is also changed and no coding is required.

Using this prototype, the most representative heuristics proposed in related work were implemented and compared with DBOwlizer. Outcomes of this comparison are described in the following chapter.

# Chapter 6 Evaluation and Comparison with Related Work

In previous chapters we have described how the DBOwlizer framework can be used to i) represent, ii) automatically generate and iii) execute mappings from relational databases to RDF/OWL ontologies using semantic web languages (*i.e.* OWL, SWRL) and semantic web technologies (*e.g.* the OWL API, reasoners).

Two of the earliest approaches to extract the conceptual model behind a relational database are the reverse engineering techniques presented in (Navathe and Awong 1987) and (Chiang, Barron et al. 1994). These approaches focus on classifying database tables based on their composition, one of the main tasks in DBOwlizer. Although their target is to obtain an extended entity-relationship model, this model shares constructs with ontologies (Section 2.3). Thus, these two works are implemented and compared as related work.

Efforts in the ontology learning community include methodologies and rule-based heuristics that focus on the interoperability of databases and ontology management applications such as the exchange of data between existing ontologies and databases. The work by Shen *et al.* (Shen, Huang et al. 2006), Astrova *et al.* (Astrova, Korda et al.

2007), Man *et al.* (Man, Xiao-Yong et al. 2005) and Stojanovic *et al.* (Stojanovic, Stojanovic et al. 2002) characterizes these efforts and are also included as related work. More recent approaches include tools and frameworks to import data from databases into existing ontologies based on manually created mappings, automatically generate ontologies from a database or use a database to store large amounts of ontology instances. These applications include DB2OWL (Cullot, Ghawi et al. 2007), DataMaster (Nyulas, Oconnor et al. 2007), D2RServer (Bizer 2003) and ROSEX (Curino, Orsi et al. 2009). The representation of mappings in these tools/frameworks are generally application dependent, and the automatically generated ontologies are not very expressive but solve the problem of data exchange between relational databases and ontologies. These tools/frameworks are also included as related work.

The existence of such heterogeneous approaches and methodologies has resulted in the need for a standard representation for data exchange between databases and ontologies. This is the main goal of the RDB2RDF working group, to provide a relational database to RDF mapping language (R2RML) to describe customized mappings (Ezzat and Hausenblas 2009). This language is a work in progress at the time of completion of this thesis. However, it promises the standardization of tools and methodologies towards the exchange of data between databases and RDF. A default mapping (RDB2RDF working group 2010) to automatically extract mappings from a relational database is also proposed by this group and included in this chapter for comparison with DBOwlizer default mapping.

In the following sections we describe the mapping heuristics used by these related approaches which are clustered in the groups: i) reverse engineering approaches, ii) heuristics/methodologies to create mappings from relational databases to OWL and ii) tools and frameworks. For comparison and evaluation purposes we have represented and executed the heuristic rules of these approaches. The names used in this section to denote a related approach, such as Stojanovic *et al.*, are also used in the rest of the document to refer to such an approach.

A summary of the OWL constructs they create is presented in Table 11. We executed each one of these mapping heuristics with our test case (employees database) as an input. Metrics of the ontologies obtained with these tests are included Table 12. The output of DBOwlizer with default mapping heuristics is also included.

## 6.1 Reverse Engineering Techniques

The following techniques are two of the most representative works in reverse engineering to obtain entities and their relations from a database. They both generate an extended entity-relationship model as output.

### 6.1.1 Navathe & Awong

Navathe & Awong propose one of the first methodologies to extract an entity-category-relation model (ECR) from relational models. ECR extends the entity-relationship model with hierarchies of classes (categories) that group similar entities. Their methodology consists of reversing the rules for the mapping from the entity-relationship to the relational model by means of heuristics involving querying the data stored in the

database and manual input to discover functional dependencies between attributes. The input to this methodology is a database in Third Normal Form and assumes a consistent naming of attributes. The first step of their methodology is to classify the database relations (tables) in the categories listed in Table 9.

Table 9. Categories of database relations and attributes used by Navathe and Awong, their corresponding classes in our relational-model ontology and the entity-category-relationship (ECR) model components created from each category.

| **Category in Navathe and Awong methodology.** | **Corresponding relational-model ontology class.** | **Corresponding ECR mapping and OWL construct.** |
|---|---|---|
| Primary relation type 1 (PR1) | 'Database entity relation' | Entity (OWL Class) |
| Primary relation type 2 (PR2) | 'Database unary relationship relation', 'Database partially unary relationship relation' | Weak entity (OWL class and object property) |
| Secondary relation type 1 (SR1) | 'Database n-ary relationship relation', 'Database binary relationship relation' | Relationship between entities (OWL class and object properties) |
| Secondary relation type 2 (SR2) | 'Database partially n-ary relationship relation', 'Database partially binary relationship relation' | Entities and relations (OWL class and object property) |
| Foreign Key attribute (FKA) | 'Database non-primary and foreign key attribute' | Relation (object property) |
| Primary Key Attribute (KAP) | 'Database primary and foreign key attribute' | Relation (object property) |
| Primary Attribute General (KAG) | 'Database primary and non-foreign key attribute' | Attribute (data property) |
| Non-key attribute (NKA) | 'Database non-key attribute' | Attribute (data property) |

Based on the categories listed in Table 9, Navathe and Awong's methodology creates their corresponding entity-category-relationship elements. This is one of the first algorithms to approach reverse engineering by using the primary keys and foreign keys of a table. The assumption of having consistent names (*i.e.* the same name for the same attribute across tables) is strong, but could be eliminated by considering that attributes referring to each other are denoted with a foreign key.

**Implementation in DBOwlizer**

The relational-to-ontology-mapping-NavatheAwong ontology contains the rule-based heuristics of this methodology. Each category created by Navathe and Awong is assigned its corresponding class in the relational-model ontology (see Table 9). Based on this classification, mappings to OWL constructs that better represent the targeted entity-category-relationship constructs are created and executed. The heuristics that include manual tasks performed by a user are not included in this description. These manual tasks include: the generation of hierarchies and the renaming of attributes or substitution of candidate keys as primary keys. Note that the output ontology reflects as much as possible the entity-category-relationship model by using the common elements of this model and OWL.

### 6.1.2 Chiang *et al.*

Chiang *et al.* extends Navathe & Awong's terminology to describe and classify relations and attributes in the relational model. The input to this method is also a database whose tables are on Third Normal Form with a consistent naming of key attributes and no error in values of key attributes to discover extra inclusion dependencies. This methodology requires users to identify to which relation a non-key attribute belongs to. Another step in this methodology requiring user input is the decision of classifying a relation as a weak entity of subclass relation. Table 10 contains the categories of relations created by Chiang *et al.*, their corresponding classes in the relational-model

ontology along with their corresponding extended entity-relationship model construct according to their heuristics.

The target of this methodology is to create an extended entity-relationship model, described in section 2.1.8. Therefore, relation characteristics such as whether a relation is transitive or functional are not part of the heuristics but cardinality constraints are included. The classification of relations and attributes is very granular and is based on the best practices to map from an entity-relationship model to the relational model.

Table 10. Categories of database relations used by Chiang *et al.,* their corresponding classes in our relational-model ontology and the extended entity-relationship model (EER) components created from each category.

| **Category in Chiang *et al.'s* methodology.** | **Corresponding relational-model ontology class.** | **Corresponding EER mapping** |
|---|---|---|
| Strong entity relation | 'Database entity relation' | Entity |
| Weak entity relation | 'Database partially unary relationship relation',<br>'Database partially binary relationship relation',<br>'Database partially n-ary relationship relation' | Entity and relation |
| Regular relationship relations | 'Database n-ary relationship relation',<br>'Database binary relationship relation' | Relationship between entities |
| Foreign Key attribute | 'Foreign key attribute' | Relation |
| Non-key attribute | 'Non-key attribute' | Attribute |

**Implementation in DBOwlizer**

The `relational-to-ontology-mapping-Chiang` ontology contains the classification of attributes (columns), relations (tables) and their mappings to OWL constructs that better represent the extended entity-relationship constructs targeted in the methodology. Some of the classes in the `relational-model` ontology are inspired by Chiang *et al.*'s classification and therefore are directly mapped to this ontology. In addition, the `relational-model-chiang-primitive` ontology contains classes not

reused in the relational-model ontology, which are included for reference. Our automated version of this methodology is to assign non-key attributes to the relation they belong to instead of ask for user's input. Similarly, our approach to automate the process of classifying a table classified as a 'Database single dependent primary key' as a weak entity or a subclass is to treat this type of table as a weak entity by default given that a foreign key implies a dependency between two entities in a more general way that a subclass relation. For instance, the table **dependent** and **manager** reference **employee** with their primary key. While manager is clearly a subclass of employee (*i.e.* all managers are employees), dependent is not (*i.e.* dependents are not employees). This example illustrates the need of user's input and our automatic version takes the conservative approach of creating an object property to represent the relation between the two classes without committing to the subclass relations. In this methodology, generalization hierarchies are created for classes that are complete covered by their subclasses, and are defined as the union of their subclasses. Additional inclusion dependencies are calculated by querying the records of the database, this process was not implemented in our prototype and therefore we cannot obtain these dependencies and generalization hierarchies were not created.

We reused the criteria of Chiang *et al.*'s classification as a basis for the relational-model ontology.

## 6.2 Heuristics/Methodologies for Mapping Relational Databases to OWL

The second group of mapping heuristics represented and executed in DBOwlizer includes methodologies and heuristics to obtain an RDF or OWL ontology from a database. The rules of these heuristics were implemented in DBOwlizer in a straightforward manner, except when ambiguous rules were described without a formal representation.

### 6.2.1 Stojanovic *et al.*

The approach presented by Stojanovic *et al.* is one of the first to map information in relational databases to ontologies. Ontologies are first created using Frame logics (Kifer, Lausen et al. 1995), which is very similar to Description Logics, and are subsequently translated to RDFS. In the mapping process, the composition of tables and keys is considered.

By default, all database tables are mapped to concepts (OWL classes) with the exception of those representing binary relations. Binary relations are mapped to a pair of two inverse properties. Tables whose primary keys are equal can either be treated as information distributed in more than one table and mapped to the same class or be mapped to two different classes related by the subclass relation. This decision is left to the user.

Columns composing a foreign key are mapped to object properties and the remaining ones to data properties.

Classes mapped from n-ary relations include an existential restriction to represent the dependency of the n-ary relation from its participants. Further constraints are imposed on columns with restricted values such as unique values.

The data migration (ontology population) is executed on the fly using a Frame Logic inference engine (Ontobroker), which is an advantage of this methodology with respect to scalability. Instances are created with new identifiers, not using primary keys or any other information from the DB identifiers. The set of simple rules presented is a first step towards the automatic extraction of ontologies from databases, however it requires user's input and does not extract complex constraints, given that the resulting language is Frames Logic and ultimately RDFS.

**Implementation in DBOwlizer**

The relational-to-ontology-mapping-Stojanovic contains the rules preivously defined. For example, Stojanovic's definition of binary relations is represented in the relational-model ontology as instances of the class 'Database binary relationship relation' that not('has part' some 'Database non-key attribute'). In our automatic approach we assert by default the subclass relation to avoid user's input when deciding between mappings into the same class or the subclass relation. A subclass relation may better preserve the distinctive information represented in the different tables. Consider for example the table **manager** in the **Employees** database, which is asserted as a subclass of **employee**. If both tables are mapped to the same

class, say, **employee,** the values for the attributes of **manager** will be empty for those employees who are not managers.

### 6.2.2 Man *et al.*

According to this methodology, classes are created from tables that have their primary key composed by one column or composed by at least one column that is not part of a foreign key. Note that in this definition binary relationships that include a non-key attribute are excluded as well as tables with composite primary keys. For example, the table **outsourceworkers** in the employees database is not mapped to the output ontology. Tables representing n-ary relationships, whose primary key is fully composed by columns that are part of foreign keys, are decomposed into binary relations. In this decomposition, semantics about the relationship are not mapped. For instance, in the **Employees** database, the ternary relation between the tables **project**, **job** and **employee** represented in the table **projectjobemployee** may have more than one binary relation between a specific project and a job which can only be identified by the employee that participates in that particular project under the particular job. This information is not mapped when we decompose n-ary relations into the binary relations between the tables: **project** and **job, job** and **employee, project** and **employee**. Finally, tables with primary and foreign keys that reference each other are mapped into the same class. The rationale behind this is that they contain information about the same entity distributed in the two tables.

Object properties are created from columns that are part of a foreign key but not part of the primary key. Primary keys of tables with a primary key that contain at least one column that is not part of a foreign key and columns that are part of more than two foreign keys are mapped into an object property named 'has part' object property. According to this methodology, the referred table of the foreign key is the domain of the 'has part', and the referring table is the range. For instance, the column **projectjobemployee.employee_id** will be mapped to the property 'has part' with employee as domain and **projectjobemployee** as range. This is somehow counterintuitive since we would consider the referring table to be the one having the component. For example, the n-ary relationship **projectjobemployee** has the employee as a component (more precisely as a participant) and not the opposite. Despite this discussion, this is the only methodology creating the 'has part' property which implies aggregation, a strong semantics commitment not implemented in our prototype.

Similarly to previous methodologies, tables representing a binary relation are mapped to a pair of inverse object properties.

Columns that are not mapped to object properties are mapped to data properties.

Cardinality constraints are created to represent restriction on the values that a column can hold such as not null or unique.

Similar to other methodologies in this group, a prototype (SOAM) was mentioned in the manuscript but not publicly available at the time of writing this thesis. As previously discussed, this methodology also considers the number of columns composing a table and keys instead of their nature, which can lead to an unintended classification or not

reflecting all the semantics in the database in the output ontology, which is not very expressive. However, it is the first methodology to propose the mapping to the 'has part' property, a building block for mereologies.

**Implementation in DBOwlizer**

Man *et al.*'s methodology to extract ontologies from databases is represented in the relational-to-ontology-mapping-Man ontology. The heuristic rules in this methodology are mapped straightforward into the definition of mapping classes and their preconditions include the composition of tables in terms of columns and keys and also the number of columns. For example, n-ary relationships that are mapped to OWL classes are those classified as instances of the class 'Database partially n-ary relationship relation'. This methodology offers the option to create a subclass relation instead of the 'has part' relation but the criteria is ambiguous and therefore not reflected in the implementation. The lack of a formal representation for the rule heuristics impeded a more precise representation of some rules in our framework. The corresponding ontology however aims to represent as many rules as possible.

### 6.2.3 Shen *et al.*

In this approach, only metadata (database schema) is mapped to ontologies, while the database instances (records) reside in the database and are retrieved *ad-hoc* for data exchange. This is an advantage with respect to scalability of this approach.

Similar to DBOwlizer, tables are classified based on their composition. However, their heuristics consider the number of attributes, which can erroneously classify tables that

have composite keys. For example, tables whose primary key contains more than one column does not fall into any rules and are not included in the output ontology. The primary key of the table **dependent** contains two columns and therefore is not mapped to a class. Similarly, tables representing n-ary relations are excluded.

This methodology maps into an OWL class tables that contain a primary key with one column. When two tables have the same primary key and a foreign key exists between them, they are both mapped to the same class unless a class has already been created for these tables, in which case a subclass relation is asserted.

Datatypes and object properties are created using the composition of keys. Domain and range for the output properties are asserted. Again, for the generation of properties composite keys are not considered and the information in columns of composite keys is not mapped such as the table oursourceworkers. Characteristics of properties such as transitivity or functionality are manually asserted by users.

Universal quantified and cardinality restrictions are automatically asserted for columns that compose keys or have value restrictions such as **UNIQUE** or **NOT NULL**. Shen *et al.* suggest the generation of additional restrictions by a user.

The generation of individuals on demand is not detailed in the manuscript.

The main advantage of this methodology is the population of the ontology on demand for query answering. A significant disadvantage is that heuristics are based on the number of columns composing keys and tables as previously discussed.

**Implementation in DBOwlizer**

The relational-to-ontology-mapping-Shen ontology contains the heuristics to extract OWL ontologies from databases with the exception of rules that require user's input, such as choosing between mappings into a subclass or the same class. This case is also presented in Stojanovic's methodology and our approach to automate the process is also to use the subclass relation by default.

### 6.2.4 Astrova *et al.*

This methodology has been widely referenced by subsequent approaches proposing heuristics for mapping relational databases to ontologies. It includes rule-based transformations of relational databases to create OWL ontologies. Similar to Stojanovic *et al.*, Astrova *et al.* 's methodology generates classes from all tables, except those that represent a binary relation, which are mapped to a pair of inverse object properties. Characteristics of properties are also asserted. For instance, properties generated from attributes in primary key are defined as inverse functional. Properties created from non-primary foreign keys that reference the same table are asserted as symmetric. However this bidirectionality is debatable and inverse properties may be more appropriate. For instance, suppose that Joe and Mary are instances of Person. The statement Joe 'has dependent' Mary does not imply that Mary 'has dependent' Joe, but an inverse property such as 'is dependent of' holds, *i.e.* Mary 'is dependent of' Joe. Along the same lines, if the object property previously described was obtained from a foreign key with the restriction on delete cascade is asserted as transitive though the methodology

does not check that the statements derived from the transitivity of the property hold in the database. For example, consider that Training, Workshop and Courses are instances of the class Project. If we assume that the property 'has subproject' is transitive and the statements Training 'has subproject' Workshop and Workshop 'has subproject' Courses, the statement Training 'has subproject' Courses has to be mapped from the database as well.

Constraints are added to properties and corresponding classes generated from primary keys. Universal restrictions i.e. *'All values from'* are created. However, without the corresponding existential restriction i.e. *'Some values from'*, this restriction can be satisfied in the case when a value for the property is not known. This restriction does not fully satisfy the semantics of a primary key, which in addition of having a restricted value, it should always have a value that identifies the row.

A limitation in this methodology is that composite primary keys are not considered and therefore are not included in the output ontology.

While Astrova *et al.* presents QUALEG DB, an application to apply the heuristics presented, this implementation was not publicly available at the time of writing this thesis and rules that could be clarified by executing this application were not applied.

As discussed before, this methodology can be improved to include a wider variety of tables and columns in databases. Nevertheless, simple OWL ontologies mapped are automatically generated from the database schema.

**Implementation in DBOwlizer**

These rules are represented in the ontology `relational-to-ontology-mapping-Astrova`. The definition of the mapping classes is created directly from the rules described in the reference manuscript, with the exception of a lexical overlap measure which requires numeric calculations not implemented in our prototype. In our implementation we only assert object properties as inverse since OWL data properties cannot be asserted as such.

The textual description of the rules instead of a more formal representation is ambiguous in some cases, such as the generation of object properties for subclasses for which only an example is provided. These rules are not included in the implementation of this methodology.

Applications and frameworks that also generate OWL ontologies automatically from a database are described in the following section.

## 6.3 Tools/Frameworks

Off the shelf tools and frameworks to automatically generate and execute mappings from relational databases to OWL are included in this section. In general, these applications follow the pattern table-to-class (*i.e.* tables are mapped to classes) and column-to-property (*i.e.* columns are mapped to properties) (Cullot, Ghawi et al. 2007) with few exceptions that consider further constraints imposed on the database. The heuristics for the generation of mappings in these applications are either encoded in the implementation or generated in a file using an application dependent markup language.

### 6.3.1 DB2OWL

Although DB2OWL is intended to map databases to already existing ontologies, heuristics represented with the $R_2O$ language (Barrasa, Corcho et al. 2004) to automatically generate mappings from databases to OWL are presented in (Cullot, Ghawi et al. 2007).

In DB2OWL, tables are mapped to classes and columns are mapped to properties with the exception of tables that represent a binary relationship. Database tables are classified in 3 categories: i) tables that relate two tables, which are mapped to two inverse object properties, ii) tables that reference another table via a primary foreign key for which a subclass relation is made and iii) tables that do not fall into the previous categories, which are mapped to OWL classes.

Object properties are mapped from foreign keys and asserted as functional while columns that are not part of keys are mapped to data properties. No further restrictions or characteristics of properties are generated.

The generation of individuals is on demand using the $R_20$ framework which provides an interpreter for the mappings to be executed. $R_20$ is an XML-based language that allows the description of rule-style mappings between relational database elements, such as tables and columns, to ontology constructs, such as classes and properties, along with conditions and operations. Conditions and operations allow the description of the prerequisites and transformations needed to create an ontology instance from a database row and their corresponding attributes. The main difference of $R_2O$ framework with respect to DBOwlizer is that the former is intended for mapping independently

conceived, developed and maintained ontologies and databases, *i.e.* not for ontologies created from the database. The strength of $R_2O$ is the expressivity of the mapping of database records and ontology instances (excluding classes), their relations and constraints. $R_2O$ describes preconditions of mappings based on a selection of rows, similar to queries in views, which are more expressive. Similarly, syntactical transformation of database instances are hard coded in our application, but explicitly described in $R_2O$.

While these rules generate simple ontologies they have the advantage that the population of the output ontology is generated on demand.

**Implementation in DBOwlizer**

The classification of tables in these heuristics are clearly explained, and led to a straightforward design of the relational-to-ontology-mapping-db2owl ontology. In this ontology, tables that relate two tables are described as 'Database binary relationship relation' that not ('has part' some 'Database non-key attribute')' and tables that reference another table via a primary foreign key are classified as a 'Database specific relationship relation'.

### 6.3.2 DataMaster

DataMaster is a plugin for the Protégé ontology editor version 3 (Noy, Crubezy et al. 2003), which allows to import data from databases to ontologies. The ontology generated by DataMaster includes classes that represent the tables, object and data properties, but also includes application dependent classes and properties.

DataMaster maps the database tables and views into classes. It also generates the following application dependent classes: Class, DatatypeProperty, ForeignKey, ObjectProperty.

Object properties are created for foreign keys that are not part of the primary key. In addition, two inverse object properties are created from tables representing binary relationship relations as defined in previous methodologies. An object property is named after the class representing the table referred by the corresponding foreign key with the postfix _INSTANCES (*e.g.* department_INSTANCES), therefore we cannot track the specific table that generated that relation if more than one foreign key references the same table. Domain and range constraints are imposed on these object properties based on the tables they belong to and reference. Application object properties, asserted as functional, are also included: hasLocFieldProperty, hasLocTableClass, hasForeignKeys, hasRefFieldProperty, hasRefTableClass.

Data properties are mapped by DataMaster from every column, even for those that have been mapped as an object property. Domain for each property is the table that the column belongs to and the range the datatype of the column. Additionally, it also creates methodology dependent datatypes: hasFKName, hasOrigColumnName, hasPrimaryKeyFields, hasReferenceField, hasReferenceTable and isBridgeTable. Ranges for data properties correspond to the data type of the column.

Individuals are created for every row of a table, named with the name of the table plus an auto generated number. Values of each column are linked through the corresponding data property.

As previously mentioned, application dependent classes and properties are included in the output ontology to represent the database schema. This representation is confusing given that a user would expect the output ontology to represent the information of the specific domain stored in the database and not information of how it is stored. This ready to use approach provides a good translation of the data where most of the relations between the data hold but the semantics about these relations are lost, such as those contained in foreign keys that are not mapped to the output ontology.

DataMaster also generates a proliferation of individuals that represent the same entity. New individuals and classes are created from views, rather than related as instances of the classes generated from the queries defining the views. For example, a new individual is created for each employee retrieved in the **high_salary_employee** view that retrieves employees with salaries equal or higher than 80,000, instead of creating a subclass of employees and using the corresponding instances of the class Employee. Likewise, columns mapped as object properties are also mapped to data properties. For tables representing binary relationship relations, the foreign keys are mapped into object properties and the table is also mapped into a class. In summary, the same database elements are mapped into more than one OWL construct and/or individual.

Given that this application is included as a Protégé plugin, this approach has been widely used for the mapping of data into RDF/OWL although modifications to the mapping heuristics would require coding skills. However, this is a good solution for obtaining linked data which is not necessarily related to expressive ontologies.

**Implementation in DBOwlizer**

The heuristics used by DataMaster are represented in the ontology relational-to-ontology-mapping-dataMaster. To emulate the DataMaster application, we add into the output ontology the application dependent classes (*e.g.* Class and ObjectProperty), the data properties (such as hasReferenceTable) and object properties (*e.g.* hasRefTableClass). Similarly, we represent the mappings to generate object properties from foreign keys, but we omitted the postfix _INSTANCES added in all the foreign key names created by DataMaster. Given the lack of documentation on the heuristics used by DataMaster, there could be alternative ways to represent them and obtain the same result (*i.e.* the output ontology). Likewise, DataMaster does not provide an explicit classification for tables, but for the sake of comparison, we represent the classification implicitly provided in the preconditions of the mappings. These heuristics may be encoded in the application code but we didn't examine the source code. Note that the output ontology generated by the Protégé plugin itself is also provided with our output ontologies.

### 6.3.3 D2RServer

D2RServer auto-generates mappings from a relational database to RDF ontologies in the D2R language (Bizer 2003). Similar to DataMaster, the mappings from D2RServer can be defined as table-to-class and column-to-property, with the exception of tables that represent binary relationships, which are mapped to a pair of inverse object properties. Object properties are created from foreign key columns, and data properties are

created from the remaining columns. No further restrictions or characteristics on the properties are created.

D2R is a declarative, XML-based language to describe mappings between relational databases into RDF. One of the main differences of D2R with respect to DBOwlizer approach is derived from the target ontology language, which is RDF to D2R and OWL for DBOwlizer. Since RDF is less expressive than OWL, the mappings to RDF ontologies are simpler than those required to OWL ontologies. D2R focuses on the mapping of linked instances, which can be created using syntactic transformation (*i.e.* patterns) while DBOwlizer focuses on the description of the mappings of classes and individuals, although DBOwlizer's individual mappings are coded in the application prototype while they are explicit in D2R.

**Implementation in DBOwlizer**

The `relational-ontology-mappping-d2rServer` ontology captures D2RServer's mappings as well as the implicit classification of the relations required to generate the mappings that are encoded in the application.

### 6.3.4 ROSEX

The system architecture of ROSEX, described in (Curino, Orsi et al. 2009), is very similar to DBOwlizer. This application extracts OWL ontologies from relational database schemas and allows query answering of the database in terms of the extracted ontology in SPARQL. SPARQL queries are transformed to SQL queries on the fly. Similar to DBOwlizer, the database schema is represented using ontologies, more specifically an

extension of the Relational.OWL (Laborda and Conrad 2005). This ontology is composed of 4 classes (column, database, primary key and table), 5 object properties (has, hasColumn, hasTable, isIdentifiedBy and references) and 2 data properties (length and scale). Although Relational.OWL is not a large ontology, individuals are used as classes, making this ontology OWL Full (for which decidable inference algorithms are not available). Similar to DataMaster, the ontology extraction heuristics are encoded in the ROSEX application and modifications to these mappings will require coding skills that some users may not possess. An interesting feature of this work is the SPARQL to SQL mappings, which include a query enrichment using the hierarchies in the ontology representing the database schema.

Like most of the methodologies presented in this chapter, the classification of tables in ROSEX takes into account their primary keys and their relationships with other tables through foreign keys. Classes are created from tables that contain at least one column that is part of the primary key but not part of a foreign key. In addition, classes are created for n-ary relationship relations, which cannot be directly translated into a (binary) role and are reified by means of a new concept representing the association. A new property is generated for each column in the table. Object properties are created from tables representing binary relationship relations as described in other methodologies, however they are not asserted as inverse of each other. The classification of binary relationships involves the number of attributes, specifically, the tables should contain two columns. This criterion excludes composite keys as discussed in previous methodologies. Object properties are created from columns that compose

foreign keys. Data properties are created from columns that are not mapped into object properties.

Additional restrictions or characteristics of properties are not asserted. In contrast to DBOwlizer approach, which focuses on creating expressive ontologies, ROSEX focuses on data exchange and query answering, retrieving data on demand via SPARQL queries.

**Implementation in DBOwlizer**

The ontology `relational-to-ontology-mapping-Curino` contains the heuristics previously described. Note that the classes of the ontology Relational.OWL are also included in our `relational-model` ontology, which allowed their representation straightforwardly.

### 6.3.5 Default mapping of the RDB2RDF group

The W3C's RDB2RDF group defines a default mapping from relational databases to RDF/OWL. This mapping is a database-instances-and-schema mapping, which is defined as a logic program with built-in predicates.

The first step in their mapping is to classify attributes that are part of a foreign key. Their logic program includes n-ary predicates with n>=1. This is one of the reasons that a straightforward representation of their rules is not possible in OWL and SWRL where only binary predicates are allowed. Similarly, for the classification of columns their rules implement negation as failure. For example, attributes that are not part of a foreign key are classified as non-foreign key attributes. This is a reasonable criterion as the Closed World Assumption will hold in the relational model.

The rules in this default mapping are simple. Tables representing binary relationships as described in previous methodologies are identified and mapped into object properties. The rest of the tables are mapped into classes. Object properties are created from foreign keys. Data properties are created from columns that are not mapped to object properties. Note that this methodology does not consider composite keys and therefore the semantics of these keys are not included in the output ontology. Similarly, only ternary relations are considered in their rules, excluding n-ary relations with n>=4.

The mapping of instances, denominated *database-instance-only* mapping, involves the generation of URIs containing the database rows. A stem URI , similar to a prefix, is created with the name of the database. URIs for tables, columns and rows are created by concatenating the name of each element to the stem URI, for example: employees/department/id. In addition, URIs created to represent a row include the value of the primary key is used when available, and otherwise a blank node is created.

The generation of URIs in DBOwlizer also includes the name of the database, table and the primary key value although with syntactical differences including the upper/lowercase and prefixes. Each row URI is asserted as an instance of the class mapped from the relation it belongs to. Columns that are not part of a foreign key are mapped to (data) properties. Note that the target output is RDF and therefore the distinction between data and object properties is not required. Columns composing a foreign key that are not part of a table representing a binary relationship are mapped as properties that will link the row URI of table that contains the column to the corresponding row URI of the table that the foreign key references. Finally, columns that

are part of a table representing a binary relationship are mapped to a property that relates corresponding row URIs from the tables that are involved in the relationship. Further restrictions such as domain and range or values allowed in a column are not considered.

It is expected that the R2RML language will become a standard recommendation to represent mapping from relational databases to RDF along with its default mapping. Note that this mapping focuses on the generation of linked data and not on the extraction of expressive ontologies. Therefore, our automatically generated mapping rules can be complementary to these and also be represented in R2RML.

**Implementation in DBOwlizer**

The rules of the RDB2RDF group's default mapping this program are represented in the ontology `relational-to-ontology-mapping-RDB2RDF`. Assertions to bypass the lack of the Closed World Assumption were included in this ontology. For instance, attributes that are not part of a foreign key are asserted as instances of the class `'Database non-foreign key attributes'`.

In this section, we have presented a summary of each one of the heuristics, methodologies and tools of works related to DBOwlizer. In the following section we will discuss the most common rules on these approaches, their unique characteristics and how our work improves or complements these efforts.

## 6.4 Comparison

Heuristics of each surveyed approach are represented by an OWL ontology and SWRL rules named `relational-to-ontology-mapping-methodology` where the word *methodology* is replaced by the name of the first author of the publication in which a heuristic is presented or the name of the tool or framework. The set of SWRL rules used by each one of the methodologies is included in the corresponding ontology. Rules were reused by all the methodologies when possible.

We will now list the common heuristics of these approaches along with those in which they differ. When referring to the classification of tables or columns, we will refer to class names in the `relational-model` ontology.

### 6.4.1 Mapping into OWL classes

All the approaches surveyed and implemented in DBOwlizer create OWL classes in the output ontology. In general, most of the approaches generate a class from all tables with the exception of those representing a binary relationship. A binary relationship is classified by most of the methodologies, including DBOwlizer (see C5 in Table 6), as `'Database binary relationship relation' and not('has part' some 'Database non-key attribute')` with the exception of Man *et al.* and Shen *et al.*

Most of the approaches, including DBOwlizer, agree to map into classes those tables classified as a `'Database entity relation'`, a table which represents an entity that can be identified by itself, or in other words, a table whose primary key does not contain a foreign key to another table. In contrast, Shen *et al.*'s and Man *et al.*'s methodologies

create classes from tables whose primary key contains one column. These two methodologies consider the number of columns instead of the nature of the columns for their mappings, which excludes tables with composite keys and therefore excludes them in their mappings. In addition, Shen *et al.*'s methodology excludes tables representing n-ary relationships in the mappings. The consequences of this criterion are noticeable in Table 12, where Shen *et al.* creates only 10 classes from the employees test case database as opposed to 16, the average number of classes generated by the rest of the surveyed approaches.

Tables whose primary key reference only one table, classified as 'Database partially unary relationship relation', are also mapped to classes by all approaches including DBOwlizer, except Man *et al.* and Shen *et al.* Similarly, all approaches but Man *et al.* and D2R server map tables classified as 'Database unary relationship relation'. The complete description of the tables mapped to classes by DBOwlizer can be found in the class 'Entity class mapping' (C1) in Table 6.

Tables whose primary key reference more than two tables are also mapped to classes in order to reify the n-ary relationship represented in such tables. In other words, the n-ary relationship is represented as a new class. These tables are classified as 'Database n-ary relationship relation' and 'Database partially n-ary relationship relation' in the relational-model ontology. All surveyed approaches, including DBOwlizer (see C2 Table 6), create classes from these tables (with the exception of Man *et al.* and Shen *et al.)* but differ in imposing additional restrictions. For example, Navathe and Awong map all tables with a primary key containing at least one column

that is not part of a foreign key to classes (those classes in the relational-model ontology that contain the word "partially"). DataMaster maps all the tables to classes, the large number of classes created by this methodology can be attributed to this (see Table 12).

### 6.4.2 Mapping into a class hierarchy

Mappings to a class hierarchy are only created in half of the methodologies surveyed. In general, tables whose primary key references one other table, and therefore depend on the referenced table to be identified, are mapped to a subclass relation *e.g.* with DBOwlizer's default heuristics (see C3 in Table 6), Manager is asserted as a subclass of Employee as described in (22).

These tables are classified as 'Database unary relationship relation' in DBOwlizer ontologies. Tables that are members of this class are also mapped to a subclass relation by Stojanovic *et al.*, Astrova *et al.* Man *et al.* and DB2OWL. Chiang et al.'s methodology needs input from the user to confirm the subclass relation and is therefore not considered in Table 11.

Equivalent classes are only created by DBOwlizer's default heuristics (see Table 11) where class definitions represent membership restrictions of the sets of data defined in views' SQL queries. These definitions are used to automatically populate such classes with the use of a DL reasoner (see C11 and C12 in Table 6).

### 6.4.3 Mapping into object properties

In general, object properties are generated from columns that compose a foreign key in all the methodologies.

Most of the methodologies generate two inverse object properties from relations that represent a binary relationship, such as worksfor, which preserves the intended meaning of binary relationships in relational databases. The first object property created has as domain the class created from the table referred by the first foreign key and as range the class created from the table referred by the second foreign key. An inverse object property is created for most methodologies with the exception of Navathe and Awong, DataMaster, ROSEX and RDB2RDF. DBOwlizer's mappings for binary relationships are fully described in C5, Table 6. Shen *et al.* and Man *et al.* differ with the common definition of binary relationships, described in subsection 6.4.1, by defining binary relationships as tables with exactly 2 columns that are part of two foreign keys but also generate inverse properties. Shen *et al.* also generates two inverse object properties from tables that represent binary relationships and contain non-key attributes, classified as 'Database partially binary relationship relation' in the relational-model ontology. The information contained in the non-key attributes is not preserved in the output ontology.

All methodologies, including DBOwlizer (see C4, Table 6), create an object property from a column that compose a foreign key that is not part of a primary key, classified as 'Database non-primary and foreign key attribute' and are not part of a binary relationship. Shen *et al.* also includes the number of columns in the heuristic rules,

creating object properties from columns that are part of tables with at least one column.

For all methodologies, except D2RServer, the domain of the extracted object property is asserted as the class mapped from the table that the column belongs to. The class mapped from the table that the foreign key references is asserted as the range of the property.

### 6.4.4 Mapping into object property hierarchies

Man *et al.* is the only methodology to suggest the generation of the subproperty relation instead of subclass relation when a subclass relation is to be asserted to a table that was mapped to an object property instead of a class, for example a binary relationship. However, a specific rule for this case is not provided.

In DBOwlizer, the default heuristics create property hierarchies for columns that are mapped to object properties and for which a subset of rows are generated from a view, defined in R25 in Table 6. See for instance the object property worksfor_dept2 described in (28).

### 6.4.5 Mapping into data properties

Most of the heuristics map into a data property those columns that are not part of a foreign key, with the exception of DataMaster that generates data properties from all columns, including those mapped as object properties. This explains the higher number of data properties generated by this methodology from our test database (see Table 12).

The domain of the data property is assigned as the class that the column belongs to. The range of the data property is asserted as the corresponding XML schema datatype in all the heuristics with the exception of D2RServer and reverse engineering techniques (Navathe and Awong and Chiang *et al.*) that focus in obtaining entity-relationships models where data properties are represented as attributes (see Table 3).

### 6.4.6 Mapping into restrictions

Class and property restrictions are generated by related approaches to represent the implicit meaning of relational database constructs. These restrictions are useful to preserve the restrictions imposed in the relational model.

Existential restrictions to enforce the existence of an object property relating the participants of an n-ary relation with the class that is reifying such a relation are created by Stojanovic *et al.*, ROSEX and DBOwlizer (see R5 in Table 6). In DBOwlizer's default heuristics, the mapping described in C2 in Table 6, generates a class that represents an n-ary relation with the additional existential restrictions.

Qualified universal restrictions are asserted by Astrova *et al.* and Shen *et al.* for object properties that are mapped from columns composing foreign keys. This restriction preserves the semantics of foreign keys in relational databases.

Cardinality constraints are also generated on object and data properties. All of the proposed cardinality restrictions represent restrictions holding in the relational model that can be straightforwardly translated into OWL. Note that minimum cardinality is another way to express an existential cardinality.

A minimum cardinality of one is imposed on classes with properties mapped from a column that composes a foreign or primary key or with the restriction **NOT NULL** by Man *et al.*, Shen *et al.*, Astrova *et al.* and DBOwlizer (see R7 in Table 6) to guarantee that the property generated from such columns should have at least one value. A minimum cardinality of zero is asserted for classes with properties mapped from columns that can hold **NULL** values according to Shen *et al.*

A maximum cardinality of one is proposed by Man *et al.* and Astrova *et al.* for those object properties mapped from a column that compose foreign or primary keys and in the case of Astrova *et al.* for columns with the restriction **UNIQUE**.

Object properties mapped from columns that can only hold a set of specified enumerated values are mapped to data properties with an enumerated range by Astrova *et al.* and DBOwlizer's default heuristics (see C10 in Table 6).

Notice that although universal restrictions, maximum and exact cardinality preserve the semantics of the relational model, they need the Closed World Assumption to be validated and therefore are not generated by DBOwlizer's default heuristics.

Other restrictions are applied to object properties that are mapped from columns that have restrictions on the values they can hold or the keys they compose.

Object properties are asserted as functional from columns that are part of a foreign key by DB2OWL while DBOwlizer's default heuristics add the extra restriction that the foreign key must refer to a primary key (see C6 in Table 6) to guarantee that the properties will indeed be functional. Inverse functional properties are mapped by Astrova *et al.* from columns that are part of a primary key or unique. However, it is not

specified what the range of these properties is. DBOwlizer's default heuristics add the restriction to assert as inverse functional those columns that have been previously created as object properties in C7 in Table 6. Similarly, Astrova *et al.* asserts object properties mapped from foreign keys as symmetric and transitive, a criterion that is debatable and has been discussed in Subsection 6.2.4. DBOwlizer's default heuristics assert as transitive those object properties mapped from tables representing binary relationship relations in which foreign keys refer to the same table as described in R2 in Table 6 and discussed in subsection 4.3.2. This criterion is based on the database design practice of creating a table with a relation that is transitive (used for instance for recursive queries) in a table referring to itself. This practice aims to overcome the limitation of the relational model that does not allow one to describe characteristics on the relations, such as transitivity. Note that DBOwlizer's default heuristics do not include the execution of queries to check that the transitive relation holds in the data stored in the database at the time of mapping as discussed in section 4.3.2. Similarly, for tables that have a foreign key referencing the same table, DBOwlizer's default heuristics also assert the object property as reflexive (see R3 in Table 7), which is not validated against the database records but reflects a possible intended design.

Data properties are asserted as functional by DataMaster and DBOwlizer's default heuristics (see C9 in Table 6). This restriction should hold in the database given that this is a requirement for a database normalized in First Normal Form.

### 6.4.7 Mapping into individuals

All of the approaches, with the exception of the reverse engineering category, generate individuals from database rows, either as dumps or on demand. The goal of reverse engineering techniques is to obtain a conceptualization of the database content instead of mapping the actual instances of such concepts.

Conversely, tools and frameworks, with the exception of DBOwlizer focus on the generation of data, at the cost of generating simple ontologies that do not reflect the semantics of the domain represented in the database. A proliferation of new individuals can be observed in these tools (see Table 12). Of these tools, DB2OWL and ROSEX offer the ability to retrieve the data on demand, which contributes to the scalability of these systems.

Methodologies and heuristics aiming to map databases to OWL create more expressive ontologies than the tools compared in this chapter while still mapping individuals in the ontology, mostly as a dump. Shen *et al.* are the only ones discussing the generation of data on demand while Man *et al.* do not provide a detailed explanation on how to generate the individuals.

Only in Shen *et al.* and DBOwlizer map individuals that are asserted as different in order to represent the Unique Name assumption that holds in the relational model.

### 6.4.8 Mapping from views

Only DataMaster, D2RServer and DBOwlizer map data contained in database views into classes, which explains the higher number of classes obtained by these approaches from

our test database (see Table 12). In the heuristics of DataMaster and D2RServer, views are mapped to classes and each column is mapped to a data property holding the value of the column of the populated view. This simple rule maps the data in the view, however the classes and instances generated are not linked with the rest of the database. This does not reflect the intrinsic and explicit relation with the tables that are queried to generate the view as discussed in subsection 6.3.2.

In contrast, DBOwlizer's default heuristics map SQL queries embedded in views by creating a defined class that is populated automatically and contains the semantics of the subset of data selected by the query, including their relationship with other tables (see C11 and C12 in Table 6 and R10-R25 in Table 7).

This distinctive feature of DBOwlizer maps additional semantics embedded in views or other applications containing SQL queries, which represent subsets of the entities that are of particular importance for the user and therefore for the domain being represented. A description of this mapping can be found in subsection 4.3.3.

Table 11, which is split in the following pages, contains a list of the OWL components (or the corresponding OWL component) created after mapping a database into an ontology or a conceptual model (such as the entity-relationship model) by each one of the methodologies, tools and frameworks presented in this chapter. A summary of the metrics of the ontology created from the employees database test case using the heuristics in related work and DBOwlizer's default heuristics is presented in Table 12. These metrics were obtained by loading in Protégé (ver. 4.1) the output ontologies obtained by the implementation of each approach in DBOwlizer.

In the next section, a comparison of the representation, creation and execution of mappings in related work with respect to DBOwlizer is presented.

Table 11. Comparison of the OWL components created by each one of the methodologies described in related work.

| | Feature/ Author | **Reverse Engineering techniques** | | **Heuristics/Methodologies Relational Databases to OWL** | | | | **Tools / Frameworks** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Navathe& Awong | Chiang *et al.* | Astrova *et al.* | Shen *et al.* | Man *et al.* | Stojanovic *et al.* | DB2 OWL | Data Master | D2R Server | ROSEX | RDB2 RDF | RDB Owlizer |
| Class restrictions | Class | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Class hierarchy | ✘[M] | ✘[M] | ✓ | ✘ | ✓ | ✓ | ✓ | ✘ | ✘ | ✘ | ✘ | ✓ |
| | Equivalent Class | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✓ |
| | Existential | ✘ | ✘ | ✘ | ✘ | ✘ | ✓ | ✘ | ✘ | ✘ | ✓[Q] | ✘ | ✓[Q] |
| | Universal | ✘ | ✘ | ✓[Q] | ✓[Q] | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ |
| | Min cardinality | ✘ | ✘ | ✓ | ✓ | ✓ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✓ |
| | Max cardinality | ✘ | ✘ | ✓ | ✓ | ✓ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✓ |
| | Exact cardinality | ✘ | ✘ | ✓ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ |

✓ Included in the mapping heuristics /tool .

✘ Not Included in the mapping heuristics /tool.

[Q] The restriction is qualified.

| Feature/ Author | | Reverse Engineering techniques | | Heuristics/Methodologies Relational Databases to OWL | | | | Tools / Frameworks | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Navathe & Awong | Chiang *et al.* | Astrova *et al.* | Shen *et al.* | Man *et al.* | Stojanovic *et al.* | DB2 OWL | Data Master | D2R Server | ROSEX | RDB2 RDF | RDB Owlizer |
| Property restrictions | Object property | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Data property | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Property Hierarchy | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | Property Domain | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ |
| | Property Range | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ |
| | Enumerated Range | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | Inverse property | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ |
| | Functional object property | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ |
| | Inverse functional object property | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | Reflexive object property | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | Transitive object property | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | Symmetric object property | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | Functional data property | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |

✓ Included in the mapping heuristics /tool .

✗ Not Included in the mapping heuristics /tool.

| Feature/ Author | | **Reverse Engineering techniques** | | **Heuristics/Methodologies Relational Databases to OWL** | | | | **Tools / Frameworks** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Navathe & Awong | Chiang *et al.* | Astrova *et al.* | Shen *et al.* | Man *et al.* | Stojanovic *et al.* | DB2 OWL | Data Master | D2R Server | ROSEX | RDB2 RDF | **RDB Owlizer** |
| Other mappings | Individuals | ✘ | ✘ | ✓ | ✓* | ✓α | ✓ | ✓* | ✓ | ✓ | ✓* | ✓ | ✓ |
| | AllDifferent individuals | ✘ | ✘ | ✘ | ✓ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✓ |
| | Collection | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✓ |
| | Views mapping | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✓ | ✓ | ✘ | ✘ | ✓ |
| | Upper level ontologies mapping | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✘ | ✓ |
| | Data types mapping | ✓ | ✘ | ✓ | ✓ | ✘ | ✘ | ✓ | ✓ | ✓ | ✓ | ✘ | ✓ |
| | DL *expressivity* | ALHI +(D) | ALHIN +(D) | ALHIN +(D) | ALHIN +(D) | ALHIN +(D) | ALEHIN | ALHIF (D) | ALUIF (D) | ALH (D) | ALEH (D) | ALH(D) | **SHOIN (D)** |
| | Implementation | ✓+M | ✓+M | ✓+ | ✘ | ✓+ | ✓+ | ✓+ | ✓ | ✓ | ✓+ | ✘I | ✓ |

✓ Included in the mapping heuristics /tool .
✘ Not Included in the mapping heuristics /tool.
Q The restriction is qualified.
M Requires manual intervention.
* Individuals generated on demand.
+ Implementation not publicly available.
α Not detailed explanation on how to generate the individuals.
I In progress.

Table 12. Metrics of the ontology created from the employees database using related work heuristics and DBOwlizer.

| Feature/ Author | **Reverse Engineering** | | **Heuristics Relational Databases to OWL** | | | | **Tools/ Frameworks** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Navathe & Awong | Chiang *et al.* | Astrova *et al.* | Shen *et al.* | Man *et al.* | Stoja-novic *et al.* | DB2 OWL | Data Master[i] | D2R | ROSEX | RDB2 RDF | **RDB Owlizer** |
| Class count | 16 | 16 | 16 | 10 | 15 | 16 | 15 | 29 | 23 | 15 | 15 | **46** |
| Object property count | 18 | 20 | 20 | 20 | 18 | 20 | 15 | 10 | 18 | 6 | 20 | **40** |
| Data property count | 31 | 31 | 31 | 26 | 31 | 31 | 20 | 86 | 31 | 30 | 31 | **55** |
| Individual count | 50* | 50* | 50 | 33 | 43 | 50* | 50* | 138 | 50 | 43 | 50 | **46** |

* Not included in methodology but implemented in DBOwlizer by default.

[i] Numbers are taken from the original implementation.

## 6.5 Discussion

DBOwlizer allows the representation, automatic generation and execution of mappings to create OWL ontologies from relational databases. In general, DBOwlizer improves related work with at least one of the following features: i) explicit representation of mappings using the Semantic Web languages OWL and SWRL, ii) heuristics to map into class and property hierarchies, iii) heuristics to assert characteristics to properties such as transitivity and reflexivity, iv) heuristics to map semantics encoded in database views, v) expressive output ontologies and vii) an implemented prototype for testing.

A discussion of related approaches with respect to these features is presented in the following subsections.

### 6.5.1 Representation of mappings between databases and OWL ontologies

This chapter has focused on the representation in DBOwlizer of all the heterogeneous heuristics found in reverse engineering techniques, database to OWL mapping methodologies and tools and frameworks. This representation is based on the `relational-model` and `relational-to-ontology-mapping` ontologies. Most of the heuristics are straightforwardly represented, with the exception of those that require reasoning with the Closed World Assumption, for which additional statements are asserted.

Some of the advantages of the DBOwlizer representation of heuristics are: i) representing mappings using standardized Semantic Web languages, thereby avoiding the use of application specific non-standard mapping languages, ii) facilitating the modifications to mapping heuristics, since it does not require to modify source code, iii) facilitating the reuse or integration of heuristics, since all of them are represented in the same language and their restrictions can easily be imported by other ontologies, similarly, iv) facilitating the automated manipulation of heuristics, providing not only a common test case database, but a set of ontologies that facilitate their comparison and vi) separating the domain knowledge from the model knowledge such that classes like table or record do not appear in the resulting ontology.

In related approaches, mappings are represented using application dependent languages or encoded in the application. The only approach that uses an ontology is ROSEX. Their ontology, an extension of Relational.OWL, is used to represent a database schema, however their mappings are still encoded in the application. Representing the mappings in the code requires programming skills that users may not possess.

D2RServer, DB2OWL and the RDB2RDF group offer the advantage of representing the mappings in an explicit markup language: D2R, $R_2O$ and R2RML respectively. $R_2O$ language is more expressive than D2R. Mappings in these languages can be automatically executed, although the implementation of interpreters for R2RML is a work in progress at the time of writing this thesis. In contrast, DBOwlizer represents the mappings using OWL and SWRL. Modifications to mappings in DBOwlizer only require an ontology editor such as Protégé. The multi-layer ontology design followed in our

ontologies facilitates the representation of different ontology extraction methodologies, yet keeping the same primitive classes for their comparison.

Another difference between these languages and the DBOwlizer ontology representation is that the mappings in R2RML, D2R and $R_2O$ are made for each table and in the case of $R_2O$ a potential subset of rows in a table, while in DB2OWL the mappings are applied to a set of tables that satisfy certain characteristics and are classified as specific types of relations in the `relational-model` ontology.

$R_2O$ mappings are more expressive than DBOwlizer's mappings in the sense that they can describe preconditions of mappings based on database records. Similarly, syntactic transformation of database instances are hard coded in DBOwlizer's prototype application, but explicitly described in $R_2O$.

### 6.5.2 Automatic generation of mappings between databases and OWL ontologies

All of the approaches surveyed and presented in this chapter have common criteria for generating the mappings between a database schema and an ontology. They consider the patterns in the composition of tables in terms of their column and keys, with the exception of Shen *et al.* and Man *et al.* that consider the composition of tables in terms of their number of columns. In most of the approaches, the names of the tables are not relevant, only in Chiang *et al.* and Astrova *et al.* a lexical analysis is suggested but not implemented in DBOwlizer's prototype.

Similarly, most of the approaches rely only on the database schema, and do not execute queries on the database records. A common assumption is that the database is at least in Third Normal Form and that all the inclusion dependencies are explicitly defined.

For the automatic generation of mappings, related approaches either focus on creating expressive ontologies with no instances (reverse engineering approaches) or the creation of instances of simpler ontologies (tools and frameworks). Furthermore, in the case of tools such as DataMaster the resulting ontology may represent concepts of the relational model in addition to the domain entities. This problem arises even when humans create ontologies based on databases such as characterizing an entity based on its database identifiers. DBOwlizer addresses this problem by adding the additional ontological commitment to map all of the output classes as subclasses of 'informational entities' (as defined in the SIO ontology) to explicitly assert that the entities obtained represent information about an entity and therefore do not represent the entity itself. Along the same lines, Astrova *et al.* generates additional constraints such as the exact cardinality and universal restrictions that represent restrictions imposed on the data, while in DBOwlizer these restrictions are not generated given that they cannot be verified with a DL reasoner because of the Open World Assumption.

As per performance on the generation of mappings, reverse engineering techniques are able to classify tables with finer granularity, which results in more expressive ontologies as an output. In contrast, tools and markup languages focus on more expressive mappings of database rows, which results in a proliferation of individuals with simpler ontologies. DBOwlizer's focus is the generation of expressive ontologies but also the

generation of individuals that will preserve the information stored in the database about entities and the relationships between them.

Another source of semantics in database applications are stored procedures, triggers, or pieces of code that impose additional constraints on the data (Hainaut 2002). The exact format and structure of these methods are language independent and varies from one database system to another. For instance, stored procedures can be implemented in a variety of programming languages, including SQL, Java or C, which make them harder to mine. These (external) additional constructs require the participation of a user for their interpretation and therefore are not considered by DBOwlizer, nor by any of the related approaches.

### 6.5.3 Automatic execution of mappings between databases and OWL ontologies

The execution of mappings in related approaches is mostly done via an application, with the exception of DB2OWL and D2RServer, where mappings are represented in markup languages and need an interpreter for their execution. This feature facilitates mapping reuse and interoperability of applications. In general, mappings are automatically executed with the exception of Man *et al.* and Shen *et al.* that require user input in some steps. DBOwlizer executes the mappings automatically with the use of a DL reasoner, however the creation of the output ontology is encoded in the application.

Most of the techniques and methodologies included prototypes that unfortunately were not available at the time of writing this thesis.

### 6.5.4 Output ontology

As presented in Table 12, the most expressive output ontology is generated by DBOwlizer's default heuristics, which corresponds to the $\mathcal{SHOIN}(\mathcal{D})$ description logic family, followed by the ontology obtained from Stojanovic's methodology, which correspond to the $\mathcal{ALEHIN}$ family and the ontologies obtained from tools that correspond to a $\mathcal{ALH}(\mathcal{D})$ description logic family.

Consistency of ontologies is important to ensure that the inferences drawn from them are correct. All of the output ontologies generated are consistent.

None of the approaches, including DBOwlizer, make a clear distinction between `'is a'` and `'part of'` hierarchies (Noy and Hafner 1997), which require contextual knowledge about the property in question that can only be available from a user input or a further refining via ontology mapping.

If a language such as R2RML becomes a standard recommendation, mappings created in DBOwlizer can easily be represented in this language.

Conclusions and future directions of our work are discussed in the following chapter.

# Chapter 7 Conclusions and Future Directions

Ontologies and databases were created for different purposes, and this is reflected in how these models differently represent, store and retrieve data. Nevertheless, relational databases are a rich source of information, making them good candidates for the generation and population of ontologies. By mapping relational databases to a more formal, expressive representation such as ontologies, it becomes possible to understand what kind of information the database contains, such that one can *discover* the nature of its contents. Moreover, since Semantic Web ontologies are machine understandable, formalizing the knowledge encoded with such ontologies can facilitate data integration and discovery for question answering. This has been illustrated by the three bio-ontologies presented in Chapter 3. Question answering opens the door to new possibilities in how users manage not only their data but also their increasing amount of knowledge, which is critical in many domains such as Life Sciences research and the delivery of Health Care.

Although a methodology to guide the manual generation of ontologies was proposed, many design questions need to be answered when creating ontologies. Ontology design decisions influence not only the availability of reasoning services, but also their performance. The knowledge domain required to create ontologies along with the

ability of creating customized parsers to map data from databases to populate ontologies are some of the factors that make the manual generation of ontologies a tedious task. Therefore, one of the goals of this thesis is to automatically generate domain ontologies from relational databases by analyzing their composition.

In this thesis we prove our hypothesis that an *expressive OWL ontology* can be *automatically* created using a set of rule-based heuristics represented in Semantic Web Languages over a normalized relational database. In addition, this ontology preserves the meaning of the database composition patterns as well as the information stored in a database.

Our framework DBOwlizer allows one to represent, automatically generate and execute mappings between databases and ontologies. DBOwlizer improves previous approaches by:

1. Representing mappings using standardized Semantic Web languages, thereby avoiding the use of application specific non-standard mapping languages.
2. Facilitating the modification and reuse of mapping heuristics.
3. Enabling the comparison of heterogeneous heuristics that map relational databases to ontologies.
4. Automatically generating decidable database-to-ontology mappings for more expressive ontologies.
5. Separating domain knowledge from model knowledge in the output ontology.
6. Mapping semantics from database views, including a pattern to characterize aggregation functions.

However, given that the heuristics used to automatically generate database-to-ontology mappings are based on table patterns, DBOwlizer will be unable to automatically distinguish between two different relationships if they are represented using the same table pattern. Similarly, although the most common SQL constructs used for the definition of a database are considered in DBOwlizer's default heuristics, additional constructs not covered (such as nested queries) can be explored in future work.

A future direction in the representation of mappings is to make the DBOwlizer mappings compatible with R2RML, a work in progress language proposal (at the time or writing this thesis) from the RDB2RDF W3C's working group for the representation of mappings between relational databases and RDF. Note that not all of DBOwlizer auto generated mappings will be able to be represented since they require OWL expressivity.

Future work also includes the development of a more robust and scalable framework application that would allow the generation of individuals on demand, for example via query rewriting similar to the ROSEX approach, and to implement incremental updates to the output ontologies to reflect database changes. Finally, a modest test case database was used for the comparison of ontologies. Another future project would be to execute the DBOwlizer's default heuristics over a larger database such as Saccharomyces Genome Database, to improve the yOWL ontology. It is anticipated that, with larger databases, the results will be similar in terms of the expressivity and design patterns in the ontology obtained.

In conclusion, we anticipate that the findings in this work will contribute towards bridging the gap between relational databases and the Semantic Web and ultimately to

the way that users, in particular scientists, retrieve and discover increasingly sophisticated knowledge.

# References


Aliprand, J., J. Allen, et al. (2004). *The unicode standard*, Adison-Wesley.

Aranguren, M. E. (2005). *Ontology design patterns for the formalisation of biological ontologies.* Faculty of Engineering and Biological Sciences. Manchester, University of Manchester. Master of Philosophy: 81.

Astrova, I., N. Korda, et al. (2007). *Rule-Based Transformation of SQL Relational Databases to OWL Ontologies*, Citeseer.

Baader, F. (2003). *The description logic handbook: theory, implementation, and applications*, Cambridge Univ Press.

Baader, F., C. Lutz, et al. (2007). *Is Tractable Reasoning in Extensions of the Description Logic SC Useful in Practice?* Journal of Logic, Language and Information(Special Issue on Method for Modality).

Baader, F. and W. Nutt (2003). *Basic description logics*. The description logic handbook: theory, implementation, and applications, Cambridge University Press: 43-95.

Bächle, M. and P. Kirchberg (2007). *Ruby on rails*. IEEE SOFTWARE: 105-108.

Barrasa, J., O. Corcho, et al. (2004). *R2O, an Extensible and Semantically based Database-to-Ontology Mapping language*. 2nd Workshop on Semantic Web and Databases (SWDB2004), Springer: 1069--1070.

Bechhofer, S., O. Noppens, et al. (2007) *Igniting the OWL 1.1 Touch Paper: The OWL API*. OWL Experiences and Directions (OWLED).

Berners-Lee, T. (1994). *Universal resource identifiers in WWW: a unifying syntax for the expression of names and addresses of objects on the network as used in the world-wide web*.

Berners-Lee, T. (2000). *XML and the Web*. XML World 2000. Boston.

Berners-Lee, T. (2009). *Semantic Web and Linked Data*. Retrieved 01/06/2011, from http://www.w3.org/2009/Talks/0120-campus-party-tbl/.

Berners-Lee, T., W. Hall, et al. (2006). *Computer science. Creating a science of the Web*. Science 313(5788): 769-771.

Berners-Lee, T. and J. Hendler (2001). *Publishing on the semantic web*. Nature 410(6832): 1023-1024.

Bin, H., P. Mitesh, et al. (2007). *Accessing the deep web*. Commun. ACM 50(5): 94-101.

Bizer, C. (2003). *D2R MAP - A Database to RDF Mapping Language*. Poster presentation. WWW2003.

Bizer, C., T. Heath, et al. (2009). *Linked data-the story so far*. Int. J. Semantic Web Inf. Syst. 5(3): 1-22.

Bray, T., J. Paoli, et al. (2000). *Extensible markup language (XML) 1.0*, W3C recommendation. Retrieved 1/06/2011, from http://www.w3.org/TR/2000/REC-xml-20001006.

Brian, M. (2002). *Jena: A Semantic Web Toolkit*. Internet Computing, IEEE. 06: 55-59.

Brickley, D. and R. V. Guha. (2003). *RDF Vocabulary Description Language 1.0: RDF schema*. W3C Recommendation. Retrieved 1/06/2011, from http://www.w3.org/TR/rdf-schema/.

Broekstra, J., A. Kampman, et al. (2002). *Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema*. The Semantic Web - ISWC 2002: First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002. Proceedings: 54.

Callahan, A., M. Dumontier, et al. (2011). *HyQue: evaluating hypotheses using Semantic Web technologies*. J Biomed Semantics 2 Suppl 2: S3.

Cherry, J. M., C. Adler, et al. (1998). *SGD: Saccharomyces genome database*. Nucleic Acids Research 26(1): 73.

Chiang, R. H. L., T. M. Barron, et al. (1994). *Reverse engineering of relational databases: extraction of an EER model from a relational database*. Data Knowl. Eng. 12(2): 107-142.

Clark, T., S. Martin, et al. (2004). *Globally distributed object identification for biological knowledgebases*. Brief Bioinform 5(1): 59-70.

Codd, E. F. (1970). *A relational model of data for large shared data banks*. Communications of the ACM 13(6): 377-387.

Cuenca Grau, B. (2007). *OWL 1.1. Web Ontology Language Tractable Fragments*. OWL 1.1. Editor's Draft. Retrieved 05/06/2008, from http://www.webont.org/owl/1.1/tractable.html.

Cullot, N., R. Ghawi, et al. (2007). *DB2OWL: A Tool for Automatic Database-to-Ontology Mapping*. Proceedings of the 15th Italian Symposium on Advanced Database Systems. 491--494.

Curino, C., G. Orsi, et al. (2009). *Accessing and documenting relational databases through OWL ontologies*. Flexible Query Answering Systems: 431-442.

Donini, F. M., M. Lenzerini, et al. (1996). *Reasoning in description logics*. Principles of knowledge representation: 191–236.

Dumontier, M. (2007a). *New Upper Level Ontology (NULO)*. Retrieved 1/06/2011, from http://ontology.dumontierlab.com/nulo.

Dumontier, M. (2007b). *OWL1 1.0 axioms for the Basic Relationship Ontology (BRO)*. Retrieved 1/06/2011, from http://ontology.dumontierlab.com/bro.

Dumontier, M. (2011). *Semantic Science Integrated Ontology (SIO)*. Retrieved 1/06/2011, from http://semanticscience.org/ontology/sio.owl.

Dumontier, M., M. Faizan, et al. (2008). *Modeling the Pharmacogenomics of Depression with OWL 1.1*. Semantic Web for Health Care and Life Sciences (HCLS 2008), Beijing, China.

Dumontier, M., L. Ferres, et al. (2008). *Semantic Annotation and Question Answering of Statistical Graphs*. MICAI 2008: Advances in Artificial Intelligence: 100-110.

Dumontier, M., L. Ferres, et al. (2010). *Modeling and querying graphical representations of statistical data*. Web Semantics: Science, Services and Agents on the World Wide Web 8(2-3): 241-254.

Dumontier, M. and N. Villanueva-Rosales (2007). *Three-Layer OWL Ontology Design.* Second International Workshop on Modular Ontologies (WOMO07), Whistler, Canada.

Dumontier, M. and N. Villanueva-Rosales (2008) *Modeling Life Science Knowledge with OWL 1.1*. Proceedings of OWL:Experiences and Directions DC (OWLED 2008).

Dumontier, M. and N. Villanueva-Rosales (2009). *Towards pharmacogenomics knowledge discovery with the semantic web*. Briefings in bioinformatics 10(2): 153.

Edelson, J. and H. Liu (2008). *JRuby cookbook*, O'Reilly Media.

Eisenberg, A. and J. Melton (1999). *SQL: 1999, formerly known as SQL3*. SIGMOD Rec. 28(1): 131-138.

Elmasri, R. and S. B. Navathe (2003). *Fundamentals of Database Systems, Fourth Edition*, Addison-Wesley Longman Publishing Co., Inc.

Eriksson, H. (2003). *Using JessTab to Integrate Protégé and Jess*. 18: 43-50.

Evans, W. E. and M. V. Relling (2004). *Moving towards individualized medicine with pharmacogenomics*. Nature 429(6990): 464-468.

Evren, S., P. Bijan, et al. (2007). *Pellet: A practical OWL-DL reasoner*. Web Semant. 5(2): 51-53.

Ezzat, A. and M. Hausenblas. (2009). *RDB2RDF Working Group Charter*. World Wide Web Consortium (W3C). Retrieved 1/06/2011, from http://www.w3.org/2009/08/rdb2rdf-charter.

Fallside, D. C. and P. Walmsley. (2004). *XML Schema Part 0: Primer Second Edition*. W3C Recommendation. Retrieved 1/06/2011, from http://www.w3.org/TR/xmlschema-0/.

Feldman, H. J., M. Dumontier, et al. (2005). *CO: A chemical ontology for identification of functional groups and semantic comparison of small molecules*. FEBS letters 579(21): 4685-4691.

Fellbaum, C. (1998). *WordNet: An electronic lexical database*, The MIT press.

Fowler, M. (2003). *Patterns of enterprise application architecture*, Addison-Wesley Professional.

Gene Ontology Consortium. (1999). *OWL Gene Ontology*. Retrieved 1/06/2011, from http://archive.geneontology.org/latest-termdb/go_daily-termdb.owl.gz

Grenon, P., B. Smith, et al. (2004). *Biodynamic ontology: applying BFO in the biomedical domain*. Stud Health Technol Inform 102: 20-38.

Groff, J., P. Weinberg, et al. (2010). *SQL: the complete reference*, McGraw-Hill/Osborne.

Gruber, T. R. (1993). *A translation approach to portable ontology specifications*. Knowledge acquisition 5(2): 199-220.

Guarino, N., C. A. Welty, et al. (2000). *Towards a Methodology for Ontology Based Model Engineering*. Proceedings of ECOOP-2000 Workshop on Model Engineering.

Haarslev, V. and R. Möller (2001). *Description of the RACER System and its Applications.* Proceedings International Workshop on Description Logics (DL-2001), Stanford, USA, 1.-3. August: 131-141.

Hainaut, J.-L. (2002). *Data Reverse Engineering,* LIBD - Institut d'Informatique - University of Namur.

Harris, M., J. Clark, et al. (2006). *The gene ontology (GO) project in 2006.* Nucleic Acids Res 34: D322-D326.

Hewett, M., D. E. Oliver, et al. (2002). *PharmGKB: the pharmacogenetics knowledge base.* Nucleic Acids Research 30(1): 163.

Hitzler, P., M. Krötzsch, et al. (2009). *OWL 2 Web Ontology Language Primer.* W3C Recommendation. Retrieved 1/06/2011, from http://www.w3.org/TR/owl2-primer/.

Horridge, M. and P. F. Patel-Schneider (2008). *OWL 2 web ontology language: Manchester syntax.* Working draft, W3C. Retrieved 1/06/2011, from http://www.w3.org/TR/owl2-manchester-syntax/.

Horrocks, I. (2005). *Applications of Description Logics: State of the Art and Research Challenges.* International Conference on Computational Science. Kassel, Germany: 78-90.

Horrocks, I., O. Kutz, et al. (2006). *The Even More Irresistible SROIQ.* Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR2006), AAAI Press.

Horrocks, I., P. F. Patel-Schneider, et al. (2004). *SWRL: A Semantic Web Rule Language Combining OWL and RuleML.* W3C Member Submission. Retrieved 1/06/2011, from http://www.w3.org/Submission/SWRL/.

Horrocks, I., U. Sattler, et al. (1999). *Practical Reasoning for Expressive Description Logics.* Proceedings of the 6th International Conference on Logic Programming and Automated Reasoning, Springer-Verlag.

Horrocks, I. and P. Schneider (2003). *Reducing OWL entailment to description logic satisfiability.* Journal of Web Semantics 17--29.

Hu, W. and Y. Qu (2007). *Discovering Simple Mappings Between Relational Database Schemas and Ontologies.* Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC/ASWC2007), Busan, South Korea 4825: 225-238.

Kazakov, Y. (2009). *Consequence-driven reasoning for horn SHIQ ontologies.* Proc. of IJCAI-09: 2040–2045.

Kazakov, Y., M. Krötzsch, et al. (2011). *Concurrent Classification of EL Ontologies.* 10th International Semantic Web Conference, Springer.

Kifer, M. (2008). *Rule interchange format: The framework.* Web Reasoning and Rule Systems: 1-11.

Kifer, M., G. Lausen, et al. (1995). *Logical foundations of object-oriented and frame-based languages.* Journal of the ACM (JACM) 42(4): 741-843.

Kiryakov, A., D. Ognyanov, et al. (2005). *OWLIM–a pragmatic semantic repository for OWL,* Springer.

Laborda, C. P. d. and S. Conrad (2005). *Relational.OWL: a data and schema representation format based on OWL.* Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling - Volume 43. Newcastle, New South Wales, Australia, Australian Computer Society, Inc.: 89-96.

Lerdorf, R., K. Tatroe, et al. (2006). *Programming PHP*, O'Reilly Media, Inc.

Maedche, A. and S. Staab (2001). *Ontology learning for the Semantic Web.* IEEE Intelligent Systems and Their Applications 16(2): 72-79.

Man, L., D. Xiao-Yong, et al. (2005). *Learning ontology from relational database.* Machine Learning and Cybernetics, 2005. Proceedings of 2005 International Conference on.

Manola, F. and E. Miller. (2004). *RDF Primer.* W3C Recommendation. Retrieved 1/06/2011, from http://www.w3.org/TR/rdf-primer/.

Martin, S., M. M. Hohman, et al. (2005). *The impact of Life Science Identifier on informatics data.* Drug Discov Today 10(22): 1566-1572.

McCollum, L., J. Larsen, et al. (2006). *World Wide Web Consortium.*

Motik, B. (2007). *On the properties of metamodeling in OWL.* Journal of Logic and Computation 17(4): 617.

Motik, B., B. C. Grau, et al. (2009). *OWL 2 Web Ontology Language: Profiles.* W3C Recommendation. Retrieved 1/06/2011, from http://www.w3.org/TR/owl-profiles/.

Motik, B., P. F. Patel-Schneider, et al. (2009). *OWL 2 web ontology language: Structural specification and functional-style syntax.* W3C Recommendation. Retrieved 1/06/2011, from http://www.w3.org/TR/owl2-syntax/.

Motik, B., U. Sattler, et al. (2005). *Query answering for OWL-DL with rules.* Web Semantics: Science, Services and Agents on the World Wide Web 3(1): 41-60.

National Centre for Biotechnology Information (NCBI). *PubMed.* Retrieved 1/06/2011, from http://www.ncbi.nlm.nih.gov/pubmed.

Navathe, S. B. and A. M. Awong (1987). *Abstracting relational and hierarchical data with a semantic data model*, North-Holland Publishing Co. Amsterdam, The Netherlands, The Netherlands.

Noy, N. F., M. Crubezy, et al. (2003). *Protege-2000: an open-source ontology-development and knowledge-acquisition environment.* AMIA Annu Symp Proc: 953.

Noy, N. F. and C. D. Hafner (1997). *The State of the Art in Ontology Design: A Survey and Comparative Review.* AI Magazine 18(3): 53-74.

Noy, N. F. and D. L. McGuinness (2001). *Ontology Development 101: A Guide to Creating Your First Ontology*, Stanford University School of Medicine.

Nyulas, C., M. Oconnor, et al. (2007). *DataMaster–a plug-in for importing schemas and data from relational databases into Protege.*

Prud'Hommeaux, E. and A. Seaborne. (2008). *SPARQL Query Language for RDF.* W3C Recommendation Retrieved 1/06/2011, from http://www.w3.org/TR/rdf-sparql-query/.

Raggett, D., A. Le Hors, et al. (1999). *HTML 4.01 Specification*. W3C Recommendation Retrieved 1/06/2011, from http://www.w3.org/TR/html401/.

Raven, P. H., G. B. Johnson, et al. (2002). *Biology*, McGraw HIll.

RDB2RDF working group. (2010). *Default Mapping*. Retrieved 1/06/2011, from http://www.w3.org/2001/sw/rdb2rdf/wiki/Default_Mapping.

Rector, A. L. (2003). *Modularisation of domain ontologies implemented in description logics and related formalisms including OWL*. Proceedings of the 2nd international conference on Knowledge capture. Sanibel Island, FL, USA, ACM Press.

Robu, I., V. Robu, et al. (2006). *An introduction to the Semantic Web for health sciences librarians*. J Med Libr Assoc 94(2): 198-205.

Schaerf, A. (1994). *Reasoning with individuals in concept languages*. Data & Knowledge Engineering 13(2): 141-176.

Shearer, R., B. Motik, et al. (2008). *Hermit: A highly-efficient owl reasoner*, Citeseer.

Shen, G., Z. Huang, et al. (2006). *Research on the Rules of Mapping from Relational Model to OWL* OWL: Experiences and Directions (OWLED).

Smith, B., W. Ceusters, et al. (2005). *Relations in biomedical ontologies*. Genome Biol 6(5).

Soldatova, L. and R. King (2007). *Ontology Engineering for Biological Applications*. Semantic Web: 121-137.

Stocker, M. and M. Smith (2008). *Owlgres: A Scalable OWL Reasoner*. OWL Experiences and Directions (OWLED).

Stojanovic, L., N. Stojanovic, et al. (2002). *Migrating data-intensive web sites into the semantic web*, ACM New York, NY, USA.

Thomas, E., J. Pan, et al. (2010). *Trowl: Tractable owl 2 reasoning infrastructure*. The Semantic Web: Research and Applications: 431-435.

Tsarkov, D. and I. Horrocks (2006). *FaCT++ Description Logic Reasoner: System Description*. Proc1 of the Int1 Joint Conf1 on Automated Reasoning (IJCA1 2006), Springer.

Uschold, M. (1995). *Towards a Methodology for Building Ontologies*. Workshop on Basic Ontological Issues in Knowledge Sharing in conjunction with IJCAI-95.

Villanueva-Rosales, N. and M. Dumontier (2007a). *Describing chemical functional groups in OWL-DL for the classification of chemical compounds*. OWL:Experiences and Directions (OWLED 2007), Innsbruck, Austria.

Villanueva-Rosales, N. and M. Dumontier. (2007b). *GO Slim*. Retrieved 1/06/2011, from http://ontology.dumontierlab.com/goslim.

Villanueva-Rosales, N. and M. Dumontier (2008). *yOWL: An ontology-driven knowledge base for yeast biologists*. Journal of Biomedical Informatics 41(5): 779-789.

Villanueva-Rosales, N., K. Osbahr, et al. (2007). *Towards a Semantic Knowledge Base for Yeast Biologists*. First International Workshop on Health Care and Life Sciences Data Integration for the Semantic Web (HCLS-DI 2007), Banff, Canada.

W3C. (2001). *W3C Semantic Web Activity*. 2011, from http://www.w3.org/2001/sw/.

W3C. (2004a). *OWL Web Ontology Language Guide*. from http://www.w3.org/TR/owl-guide/.

W3C. (2004b). *Semantic Web Best Practices and Deployment Working Group*. from http://www.w3.org/2001/sw/BestPractices/.

Wheeler, D. L., T. Barrett, et al. (2006). *Database resources of the national center for biotechnology information*. Nucleic Acids Research 35(suppl 1): D5.

Widenius, M., D. Axmark, et al. (2002). *MySQL reference manual*, O'Reilly & Associates, Inc.

Wikipedia. (2001). *The Free Encyclopedia*. Retrieved 01/01/2011, 2011, from www.wikipedia.org.

Winans, J. and K. H. Davis (1990). *Software Reverse Engineering from a Currently Existing IMS Database to an Entity-Relationship Model*. ER: 345-360.

Yazici, A. and Z. Karakaya (2007). *JMathNorm: A Database Normalization Tool Using Mathematica*. Computational Science – ICCS 2007: 186-193.

Zinner, M., A. Kojima, et al. (2011). *MySQL Workbench*, Sun Microsystems, Inc.