

CRISTIANE APARECIDA GONÇALVES HUVE

**AN ARCHITECTURE FOR MAPPING RELATIONAL
DATABASE TO ONTOLOGY**

Dissertation presented as partial requirement
to obtain a master's degree. Postgradu-
ate Program in Computer Science at Federal
University of Paraná.
Supervisor: Prof. Letícia Mara Peres, Ph.D.

CURITIBA

2017

CRISTIANE APARECIDA GONÇALVES HUVE

**AN ARCHITECTURE FOR MAPPING RELATIONAL
DATABASE TO ONTOLOGY**

Dissertation presented as partial requirement
to obtain a master's degree. Postgraduate
Program in Computer Science at Federal
University of Paraná.
Supervisor: Prof. Letícia Mara Peres, Ph.D.

CURITIBA

2017

H973a

Huve, Cristiane Aparecida Gonçalves

An architecture for mapping relational database to ontology / Cristiane Aparecida Gonçalves Huve. – Curitiba, 2017.

112 f ; il. color : 30 cm.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2017.

Orientador: Leticia Mara Peres

Bibliografia: p. 80-85.

1. Banco de dados relacionais. 2. Ontologia. 3. Mapeamento conceitual. I. Universidade Federal do Paraná. II. Peres, Leticia Mara. III. Título.

CDD: 005.756

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da dissertação de Mestrado de **CRISTIANE APARECIDA GONCALVES HUVE** intitulada: **An Architecture for Mapping Relational Database to Ontology**, após terem inquirido a aluna e realizado a avaliação do trabalho, são de parecer pela sua aprovação.


Curitiba, 01 de Fevereiro de 2017.


LETICIA MARA PERES

Presidente da Banca Examinadora (UFPR)


MARCOS DIDONET DEL FABRO

Avaliador Interno (UFPR)


HEGLER CORREA TISSOT
Avaliador Externo (UFPR)

(suplente)
Prof. Eduardo C. de Almeida
Deplo. de Informática
Mat.: 201461 - UFPR



ACKNOWLEDGEMENTS

I would like to thank:

My family and especially my fiancé, Giordano Dornelles, for always believing in me, by the comprehension in moments of absence and the unconditional patience.

My supervisor, Leticia Mara Peres, and Marcos Didonet Del Fabro, for the patience, the valuable guidance, and all provided support to help me in this work.

My professors, Rosane Pasarini and Jefferson Martins, and my cousins, Alline Lara and Aramis Fernandes, for the motivation and the helping in joining the graduate program.

My colleagues, Hegler Tissot and Alex Porn (FAES Research Group), for help me with ontology concepts, and Jeovane Alves, for helps me in reviewing the dissertation.

My friends who always believed it was possible and supported me at all times, especially Flávio Araujo and Jariel Luvizon.

My colleagues of the graduate program by the precious coffee.

Informatics Department of Federal University of Paraná for having provided structure to accomplish this work.

The Dental Uni Cooperative and Wise Systems for the permission to use the SIO corpus.

CAPES, which financed part of this work.

Everyone who helped me in different ways to conclude this work.

RESUMO

Nos últimos anos tem sido propostos trabalhos sobre definições de mapeamento de um banco de dados para ontologias. Este trabalho de mestrado propõe a construção de uma arquitetura que viabiliza um processo de mapeamento automático de um banco de dados relacional para uma ontologia OWL. Para isto, faz uso de regras novas e existentes e tem como contribuições a nomeação dos elementos e sua eliminação quando duplicados, aumentando a legibilidade da ontologia gerada. Destacamos na arquitetura a estrutura de mapeamento de elementos, que permite manter uma rastreabilidade de origem e destino para verificações. Para validar a arquitetura e as regras propostas, um estudo de caso é realizado utilizando um banco de dados de atendimento odontológico.

Palavras-Chave: Banco de dados relacional. Ontologia. Mapeamento.

ABSTRACT

In recent years a number of researches have been written on the topic of definitions of mapping of a database to ontology. This dissertation presents the proposal and the construction of an architecture which enables an automatic mapping process of relational database to OWL ontology. For this purpose, it makes use of new and existent rules and offers as contributions naming and elimination of duplicated elements, increasing the legibility of the generated ontology. We stand out the structure of element mapping, which allows to maintain a source-to-target traceability for verifications. Validating of proposed architecture and rules is made by a case study using a dental care database.

Key-words: Relational database. Ontology. Mapping.

CONTENTS

1	INTRODUCTION	2
1.1	Motivation	3
1.2	Objectives	3
1.3	Organization	4
2	STATE OF THE ART	5
2.1	Database design	5
2.2	Ontologies	9
2.3	Mappings	15
2.4	Related works	17
3	MAPPING ARCHITECTURE	22
3.1	Architecture components	23
3.1.1	Configuration template	24
3.1.2	Mapping model	25
3.1.3	Mapping process	28
3.1.3.1	Naming ontology elements	28
3.1.3.2	Data types	30
3.1.3.3	Mapping rules	30
3.1.4	Generation of ontology	53
3.2	Prototype	53
3.2.1	Development of mapping process	55
3.2.2	Use of mapping model	56
3.2.3	Generation of OWL ontology	57
3.2.4	Deployment diagram of architecture	57
3.3	Summary	58

4 CASE STUDY	60
4.1 Objectives	60
4.2 Scenarios	60
4.3 Method	63
4.4 Results and discussions	66
4.4.1 Rule discussions	67
4.4.2 Architecture discussions	70
4.4.3 Comparative discussion	73
4.5 Summary	75
5 CONCLUSIONS	77
5.1 Future work	79
REFERENCES	85
A STRUCTURE OF CONFIGURATION TEMPLATE	86
B ER MODEL OF MAPPING SCHEMA	87
C DETAILS OF MAPPING SCHEMA TABLES	88
D PROTOTYPE CONFIGURATION	89
E SCRIPT OF MAPPING SCHEMA	90
F CASE STUDY - DETAILS OF DATABASE TABLES	112

LIST OF FIGURES

2.1	Mapping cardinalities: a) one to one; b) one to many; c) many to one; and d) many to many. Figure of [49]	7
2.2	An example of data model representation	8
2.3	OWL ontology definition metamodel. Figure of [24]	11
2.4	Relational to ontology mapping	16
3.1	Architecture components	24
3.2	ER model - part A - mapping of database elements	26
3.3	ER model - part B - mapping of ontology elements	27
3.4	Components of mapping process	28
3.5	Architecture components of prototype	54
3.6	Prototype	55
3.7	Deployment diagram of architecture	58
4.1	Activities of method	64
4.2	Class hierarchy classification: a) asserted and b) inferred	69
4.3	Example of mapping rules	69

LIST OF TABLES

2.1	Common rules of mapping relational database to ontology	17
3.1	Correlated data types	30
4.1	Number of database elements	62
4.2	Number of ontology elements	66
4.3	Comparative of generated ontology elements	74

CHAPTER 1

INTRODUCTION

Nowadays there are different computer systems for the same business area. In the vast majority, each system models a database structure with definitions, concepts, and relations, according to specific processes practiced in the organization. Facing the definition of different models, achieving the interoperability of these systems is a great challenge to be overcome [51].

Ontologies prove to be useful in supporting the specification and development of computer systems. In computing, an ontology is defined based on a set of concepts in which a domain of specific knowledge is modeled. The definitions of these concepts include information about the semantics and restrictions applied on the domain [27]. Ontologies offer advantages in their use, such as: providing an exact description and an exact vocabulary for representation and sharing of knowledge and extending the use of a generic ontology for a specific domain [28].

From the considerations presented, we understand that the elaboration of an ontology to represent a specific domain is an alternative to handle interoperability issues. However, the process of elaborating an ontology is a task which requires a great amount of effort [51, 52]. Considering this, related works proposed building ontologies from a database of existing systems [45, 46, 60, 17, 9, 60, 13].

The main related works which propose mapping relational database to ontologies, mostly utilize the physical database model or they perform the extraction of this information directly from schema [45, 46, 60]. Others, in addition to the physical model, consider the database tuples [34, 52]. Recent works [17, 58] suggest new treatments for the transformation process, however, the logic model was not identified in any of the analyzed related works.

At the beginning of the database modeling, the definition of the model takes place,

which includes a description of the structures stored and the relations among them. When defining the physical model from the logic model, we define the structure of this model according to the database management system (DBMS).

For simplification reasons or due to DBMS restrictions, definitions are adapted and consequently lose semantic definitions initially represented in the logic model. The retrieval of this information for the ontology elaboration is of great importance to the naming of its elements. We identified a significant number of rules which establish conditions for mapping [9, 60, 13, 16, 45, 46]. However, in none of them, details are given regarding the naming of the elements, an important issue to ontology legibility. In general, the mapping process is quite abstract and there is no source-to-target relationship of the elements.

1.1 Motivation

Given the above context, the following motivations for the present study are given: a) the existing mapping processes only perform a direct mapping of the elements and do not reuse pre-existent definitions of elements; b) the existing solutions do not deal with the naming of mapped elements considering the name of the database element without formatting; c) the proposed solutions do not consider the information from the logic model in the mapping process and for naming the ontology elements; and d) in face of different contributions, we identify the opportunity to consolidate the proposed functionalities of different works in a unique architecture.

1.2 Objectives

The main objective of this work is to define an architecture that, from a set of rules, performs the mapping process from relational database elements to ontology elements. We have the following specific objectives: a) to define an architecture that supports the mapping process and the relations of elements of a relational database to an ontology; b) to identify and define a set of rules to be applied in the mapping process of the architecture; and c) to validate the proposed architecture and rules through the development of a

prototype.

1.3 Organization

Considering the facts above, we have developed an architecture to perform an automatic mapping from relational database to ontology. In this architecture, besides the database schema and tuples, its logic model is considered. The mapping process uses existing and new rules which, in addition to defining new mapping criteria, aim at the elimination of duplicated elements and the representation of concepts. A schema for mapping database elements to ontology elements has been developed.

It is intended with this architecture, to obtain an automatic mapping process, applies validations for naming elements, in order to generate an ontology with greater legibility; maintains the traceability of database elements and ontologies; and contribute to the representation and interpretation of specific domains of a database structure.

A case study, in which we elaborated three different scenarios over a dental care database, was conducted in this work with the objective of validating the proposed mapping architecture and rules.

This work is presented into five chapters. In Chapter 2 concepts and definitions of relational database and ontologies used in this work are presented. We also present concepts of mappings and the previous works which proposed mapping rules. In Chapter 3 the architecture elaborated in this research is described, justifications for its conception, the mapping rules, the details about each of the architecture components and the developed prototype. In Chapter 4 the case study is presented, in which, through three different scenarios, a relational database is transformed to ontology in Web Ontology Language (OWL). We report the details and results of the case study conducted to validate the architecture and the mapping rules and we perform a comparative discuss. In Chapter 5 we present the conclusions and proposals for future works.

CHAPTER 2

STATE OF THE ART

This chapter presents concepts review. Section 2.1 provides an introduction to a database, data modeling, and the relational model. Section 2.2 and Subsection 2.2 presents a brief introduction of ontology concepts and OWL ontology. Subsection 2.2 presents a class definition. Subsections 2.2 and 2.2 demonstrates property characteristics and definitions. Subsection 2.2 explains the concept of instances. Section 2.3 presents concepts and uses of mapping, which support our architecture and Section 2.4 presents an analysis of related works which map a relational database to ontologies.

2.1 Database design

Database management system (DBMS) is a software which enables to create, maintain, use and share large data among various users and applications [44]. Its primary goal is to provide an environment in which it is both convenient and efficient on retrieving and storing information in a database [49].

A database is a collection of related data, typically describing information about one or more related concepts [21, 44]. By data, we mean a set of arranged and interrelated information, which can be recorded and has implicit meaning [21]. A database has implicit properties: it represents aspects of the real world; it contains logically coherent data with inherent meaning; and it is designed, built, and populated with data for a specific purpose. A database can be of any size and complexity [21], but it is mostly designed to manage large bodies of information [49].

Modeling is part of database design which defines a capable structure for data manipulation. Modeling is divided into three levels: conceptual, logical and physical, which together propose to obtain a unified view of creation and maintenance of the information. At high-level, data requirements are mapped into a conceptual data model, sufficiently

detailed to describe its scope [29], and to determine the way data can be stored [45]. Next level matches the logical specification of a data model, which groups the information into structures (e.g. entities, attributes) and describes how the information will be structured (e.g. relationships, integrity rules). Last, physical data model corresponds to the internal organization of data storage. Physical data model match data manipulation and an efficient data retrieval. [29].

Entity-relationship data model (ER model), developed by Chen [14], is a conceptual data modeling tool, which corresponds a data representation of DBMS in a diagram. The ER model has three main concepts: entity, which represents an object in the real world; attributes, which are properties describing an entity; and relationships, which are association among two or more entities. Conceptual modeling uses ER model to generate entity-relationship diagrams and describes database data and their interrelationships [29, 47].

Cardinalities specify the number of entities which another entity can be associated and are most useful in describing binary relationship [49]. For a set of binary relationships between entity sets A and B, mapping cardinalities must be one of the following [48, 49]:

- **one to one:** an entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A, as it can be seen on Figure 2.1a;
- **one to many:** an entity in A is associated with any number of entities in B. An entity in B can be associated with at most one entity at A, as it can be seen on Figure 2.1b;
- **many to one:** an entity in A is associated with at most one entity in B. An entity in B can be associated with any number of entities in A, as it can be seen on Figure 2.1c;
- **many to many:** an entity in A is associated with any number of entities in B, and an entity in B is associated with any number of entities in A, as it can be seen on Figure 2.1d.

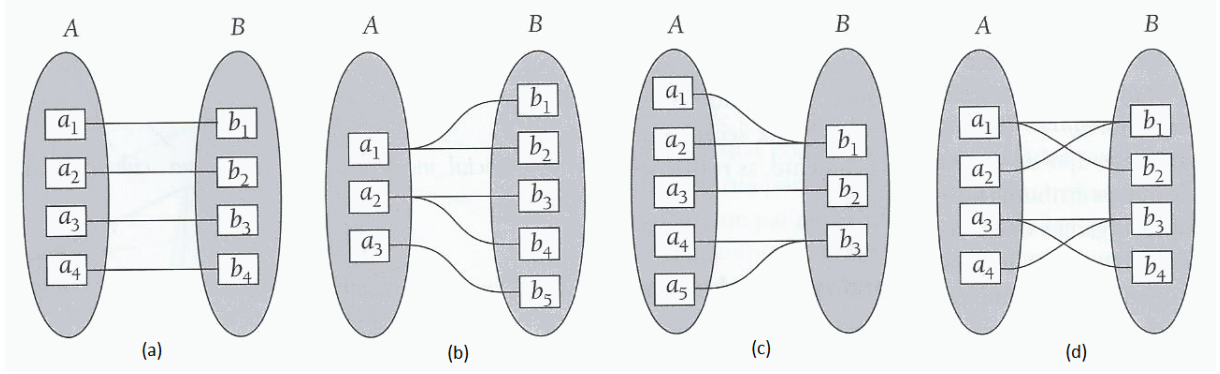


Figure 2.1: Mapping cardinalities: a) one to one; b) one to many; c) many to one; and d) many to many. Figure of [49]

Database definition involves specifying data types, structures and constraints of data to be stored. Database definitions and all descriptive information are stored by the DBMS in a database catalog or dictionary [21]. Data specification from a database is called data schema. Database schema reflects the design and the data specification of the database. A specific content of the database is called instance [29].

ER model is commonly used in commercial implementations of DBMS's, which is considered a standard in the area of conceptual methods of design and database tools [47, 49]. ER model is typically used for the representation of an initial database design in high-level. After the ER design, the model needs to be converted into a usable database. Given an ER diagram, a standard approach is taken to generate a relational database schema, which closely approximates the ER design.

Relational model

The relational data model was proposed by Codd in 1970. From ER model, entities are mapped to tables, attributes become columns, and relationship types are mapped to relations in the relational model [44].

A relational schema R , denoted by $R(A_1, A_2, \dots, A_n)$, is composed by a relation name R and a list of attributes A_1, A_2, \dots, A_n . Each attribute A_i is a name of a role played by some domain D in the relational schema R . D is called domain of A_i and is denoted by $dom(A_i)$. A relational schema is used to describe a relation; R is called a name of

this relation. A relation (or relation state) r of a relational schema $R(A_1, A_2, \dots, A_n)$, also denoted by $r(R)$, is a set of n -tuples $r = t_1, t_2, \dots, t_m$. Each n -tuple t is an ordered list of n values $t = (v_1, v_2, \dots, v_n)$, where each value v_p , $1 \leq p \leq n$, is an element of $dom(A_i)$ or is a special NULL value [21].

The database schema specifies a database structure and the database instance specifies the actual content of a database. [7]. The database schema contains tables, in the which is possible to represent domains. Table names and column names are used to understand the target domain and the content which is persisted. Figure 2.2 shows a representation of a data model, where each table (*student*, *class*, *student_class* and *country*) and each column (student [*id_student*, *name_student*, *id_country*], class [*id_class*, *description*], *student_class* [*id_student*, *id_class*] and country [*id_country*, *description*]) have an identifier name.

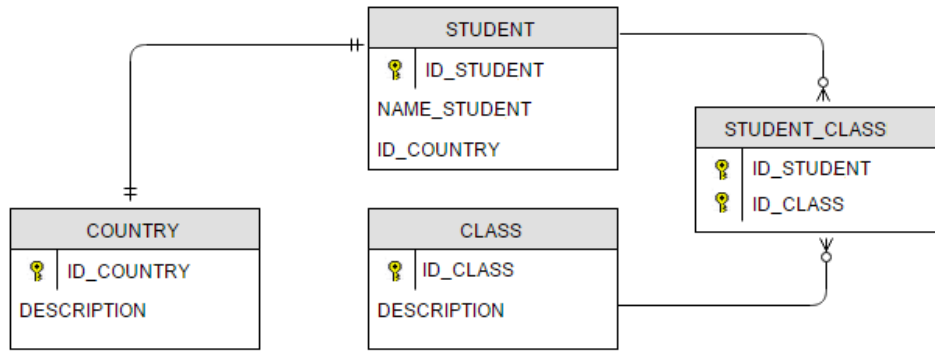


Figure 2.2: An example of data model representation

Integrity constraints are specified on a database schema and they are used to prevent the entry of incorrect information [21]. An integrity constraint is a condition which restricts data and can be stored in a database instance.

The key allows identifying stored data referring it by a unique identifier [49]. A unique key (UK) is composed of one or more columns, and it forces a unique information in each table tuple. A primary key (PK) is composed of one or more columns which, taken collectively, allows us to identify uniquely a table tuple of the database [44]. In Figure 2.2, PKs *id_student*, *id_class* and *id_country* are identified with a key representation before the column description.

In order to relate two or more tables, they must have a common relationship. An

integrity constraint involving two relations is called foreign key (FK) constraint [44]. In Figure 2.2, the data model has the associative table *student_class*, representing a multiple relation of two tables *student* and *class*. *Student_class* has two keys *id_student* and *id_class* which are inherited of source tables.

Sometimes, the data stored in a relation is linked to the data stored in another relation. This means a relation may have more than one key. Another type of foreign key relation can be represented with, e.g tables *country* and *student*, in which *student* table has a column *id_country* referring to the primary key of *country* table.

2.2 Ontologies

Ontology comes from two Greek words *ontos* (being) + *logos* (science, study) [33]. Ontology concepts are strongly associated to philosophy, as they are considered a subject which treats the being and their relations [30]. The use of the term ontology in computer science is related to build knowledge bases using automatic computational reasoning, with interoperable structures that describes concepts and relations among them [42, 30, 39].

According to Sowa [50], an ontology categorizes the existence of things and defines a set of terms to represent knowledge on a given domain. For Gruber [26], ontologies are explicit formal specifications of terms and relations among them, of a shared conceptualization of things in the domain. For “explicit specifications”, an ontology specifies an explicit and unambiguous definition of concepts; for “formal”, ontology needs to be understood and processed by computers; for “shared”, ontology needs to contain a consensual definition about a domain; and for “conceptualization”, there is an abstract model representing concepts and relations among them.

Ontologies standardize meanings through semantic identifiers, which can represent the real and conceptual world [23], using a specific and shared vocabulary to describe a domain, capturing concepts and relations, and axioms to restrict its interpretation [18].

An ontology O can be represented using five basic ontology elements $O = (C, P, I, V, A)$ where *ontos* C , P , I , V , and A are the sets of classes, properties, instances, property values and axioms, respectively [41, 59]. Classes are used to represent a group of elements,

which have similar characteristics and form a capable concept to represent an object belong to a domain. An ontology can be represented by a tree structure. This structure is composed of classes and subclass which inherit class properties which are associated. Associations are described by binary properties. Properties describe characteristics and interaction types among concepts of a domain. Axioms represent constraints which must be strictly adhered. Instances represent individuals which share class properties, and property values is a set of instances for each of these properties [30, 42].

Ontologies are developed to specify standard procedures to extract and share knowledge among different systems. They are used to ensure the consistency of system extracted data and the sharing of data among computers and humans [55].

OWL ontologies

OWL (Web Ontology Language) is a web-based language, defined by W3C (World Wide Web Consortium) to define and instantiate ontologies. OWL can be used to explicitly represent the meaning of terms in vocabularies and relations between them [1, 2].

OWL is a language developed to represent the information to be used by applications which need to process the modeled information content instead of just presenting information [35]. The main elements of OWL ontologies concern classes, properties, instances of classes, and associations between these instances. These elements are described by a rich vocabulary [24, 1].

This language allows specifying how to derive logical consequences using formal semantics, where is possible to clarify facts which are represented by semantic relations [1]. Logical expressions in OWL ontologies are defined by the representation of axioms [42], which describe relations among other ontology elements, as properties, classes and instances [56].

Model is a specification or description of a system and its environment with a particular purpose [11]. Metamodel defines possible structures and meanings for the elements of a model [20]. A model must be defined according to the structure provided by its metamodel, that is, a template must conform to its metamodel [8]. OWL Ontology Definition

Metamodel (ODM) is composed of two models to represent OWL structure, defined by RDFS (Resource Description Framework Schema) and OWL [51, 24]. RDFS is a met-language which besides representing its concepts, also defines OWL model, inheriting RDFS structure definitions [24]. Figure 2.3 presents ODM substructure.

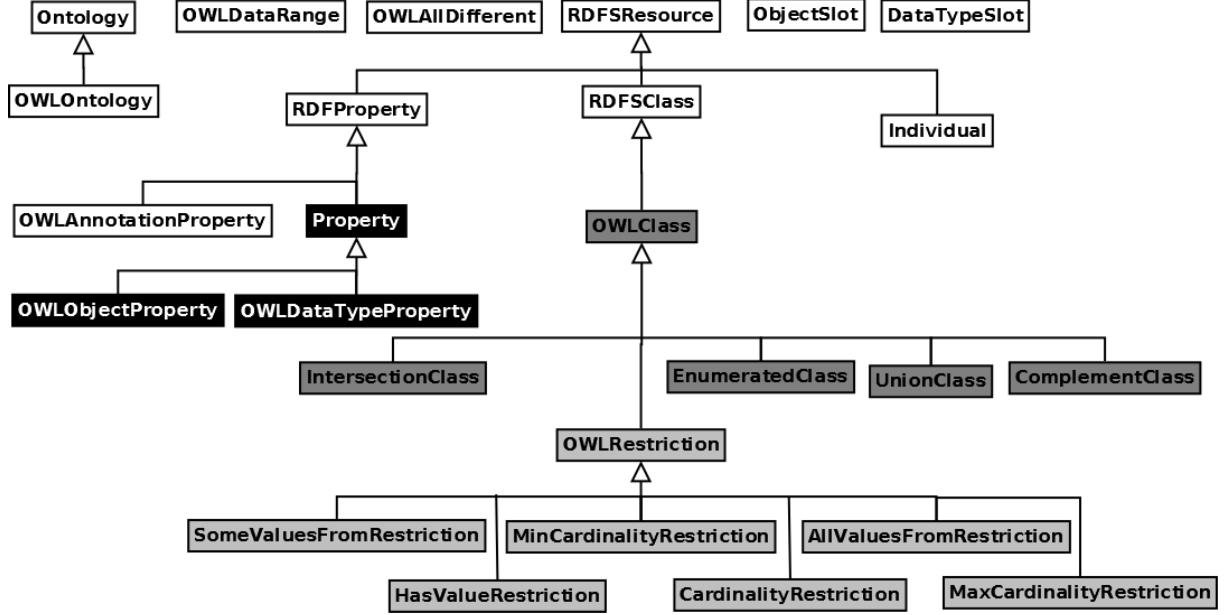


Figure 2.3: OWL ontology definition metamodel. Figure of [24]

In this work, we use definitions of OWLClass, OWLRestriction, Property and Individual, which are detailed in following sections.

Classes

Classes provide an abstraction mechanism for grouping resources with similar characteristics [1]. Class is a collection of common properties, which can be named and enumerated, and it describes a set of individuals organized in a taxonomy [41, 1]. Classes correspond to a set of things which naturally occur in represented domain [1].

Considering Figure 2.3, an owl:Class is a subclass of rdfs:Class, this inheritance allows to define hierarchical classes, and represents that a class is a subclass of another [51]. According to Example 1, *universityStudent* class is defined as a subclass of *student* class, establishing all instances of *universityStudent* class also belong to *student* class.

Example 1:

```

<owl:Class rdf:ID="universityStudent">
    <rdfs:subClassOf rdf:resource="#student" />
</owl:Class>

```

A class can be constructed in OWL as an intersection, a union or as a complement of other classes using operators *owl:intersectionOf*, *owl:unionOf* and *owl:complementOf*, equivalent to operators “AND”, “OR” and “NOT” used for descriptive logic [24, 42].

Disjointness of a set of classes can be expressed using the *owl:disjointWith* operator. Disjointness ensures an instance of a class does not have a common instance in other specific class [1]. Example 2 shows *universityStudent* class is disjoint from *masterStudent* and *doctoralStudent* classes, which can state every individual who is associated with *universityStudent* class cannot be defined as an instance of another member of listed classes.

Example 2:

```

<owl:Class rdf:ID="universityStudent">
    <owl:disjointWith rdf:resource="#masterStudent" />
    <owl:disjointWith rdf:resource="#doctoralStudent" />
</owl:Class>

```

Properties

Properties in OWL ontologies are binary associations among individuals and they allow to represent general facts about class members and specific facts about individuals [1, 26]. Considering Figure 2.3, we have two distinct types of properties: *owl:ObjectProperty*, which relates a class to another class, and *owl:DatatypeProperty*, which is a relation between class and datatype value, both organized in a taxonomy [41, 24].

Properties are characterized by their domain, range or algebraic characteristics [41]. More than one domain or range may be declared and it can be specified to restrict the relation. For each datatype property and object property, the operator *rdfs:domain* can be set. When the domain is specified, a restriction is established. Restriction represents a property belongs to a domain of one or multiple classes. OWL Datatype properties

connect an individual to a literal data type. Considering the *student* class, we can associate an integer numeric data type which represents *age*.

Example 3:

```
<owl:DatatypeProperty rdf:about="#age">
  <rdfs:domain rdf:resource="#student"/>
  <rdfs:range rdf:resource="#xsd:integer"/>
</owl:DatatypeProperty>
```

We can define a restriction of values in object property of multiple classes using operator *owl:unionOf* [3]. Example 4 presents the domain of *hasStudentBackground* object property, which can be either one or several individuals of *experience* class or *qualification* class, i.e. this property should only be used by individuals of this two classes.

Example 4:

```
<owl:ObjectProperty rdf:ID="hasStudentBackground">
  <rdfs:domain>
    <rdfs:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#experience"/>
        <owl:Class rdf:about="#qualification"/>
      </owl:unionOf>
    </rdfs:domain>
  </owl:ObjectProperty>
```

Properties can connect individuals of a domain to individuals of a range. According to Example 5, object property *hasIdentification* connects individuals belonging to *student* class to individuals belonging to *studentIdentification* class.

Example 5:

```
<owl:ObjectProperty rdf:ID="hasIdentification">
  <rdfs:domain rdf:resource="#student"/>
  <rdfs:range rdf:resource="#studentIdentification"/>
```

```
</owl:ObjectProperty>
```

Restrictions

Restriction is a concept tightly connected to properties characteristics [1]. OWL Restriction defines an anonymous class of individuals which satisfy certain restrictions on their properties [51]. Basically, it distinguishes two types of property restrictions: value constraints, which establish constraints on the range of properties, and cardinality constraints, which establish constraints on the number of values which properties can represent. Value constraints are modeled using *HasIndividualValue*, *HasLiteralValue*, *SomeValuesFrom*, and *AllValuesFrom* elements.

Cardinality constraints are modeled using *MinCardinality*, *Cardinality*, and *MaxCardinality* elements [3, 24]. Cardinality constraint establishes any instance of a class may have an arbitrary number (zero or more) of values for a particular property. In this way, *owl:maxCardinality* and *owl:minCardinality* allows specify the number of values, which can be maximum or minimum, respectively. According to Example 6, *course* class require being taught by at least someone [51].

Example 6:

```
<owl:Class rdf:ID="course">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#isTaughtBy"/>
      <owl:minCardinality rdf:datatype="xsd:nonNegativeInteger">
        1
      </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```


Individuals

Individuals or instances are used to represent specific elements in class extension, i.e. to represent data which populate ontology structure [3, 51]. Individuals should correspond to actual entities, which can be grouped into those classes [1]. An instance is presented in Example 7.

Example 7:

```
<rdf:Description rdf:ID="12345">
    <rdf:type rdf:resource="#student"/>
</rdf:Description>
```

or equivalently:

```
<student rdf:ID="12345"/>
```

Example 8 relate an instance of *age* "39" to *student_id* "12345".

Example 8:

```
<student rdf:ID="12345">
    <uni:age rdf:datatype="xsd:integer">
        39
    </uni:age>
</student>
```

2.3 Mappings

Mappings are defined to relate different types of models [19] and, in this sense, the term mapping is used in this work to refer to the relations between database elements and ontology elements. A mapping can be designed to overcome more specific or broader needs, involving one or many technologies, in different degrees of complexity and formats [19].

Through the definition of criteria, the mapping process performs transformations which tell us how an instance of a source data can be translated into an instance of a target data [38]. These criteria define the relations to be established between the models and may contain definitions regarding the naming of the elements and eventual definitions of exceptions or restrictions.

The relations indicate that certain elements of a source model $M1$ are mapped to certain elements of a target model $M2$, and can be established by a direct (1..1) or multiple cardinalities (1...N) between the elements of each model. The specification of a mapping between $M1$ to $M2$ must to support the transformation rules [19, 43].

Let us consider the two models in Figure 2.4. The first model is a relational schema $M1$ and the second model is an ontology structure $M2$. Schema $M1$ is composed of database elements (tables, columns, tuples, and relationships). Ontology $M2$ is composed of ontology elements (classes, properties, instances, and associations). The translation from $M1$ to $M2$ is represented by the mapping $M1$ to $M2$ which establish links between the model elements.

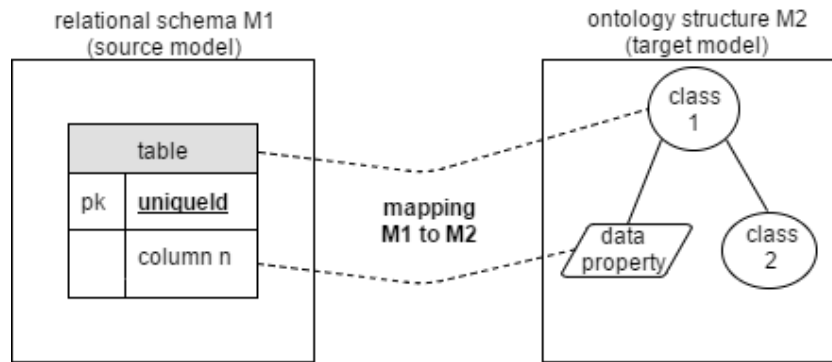


Figure 2.4: Relational to ontology mapping

In automated mapping processes, the relations can be stored permanently or kept in memory. In [19] it is recommended the creation of structures which contain specific entities to represent and store the relations established in the mapping process. This allows their use in the future in different ways, those being: for checking, modification, verification or reuse [19]. In this text, we will use the term rules instead of relations.

2.4 Related works

Tissot ¹ identified related works [9, 13, 16, 25, 32, 34, 36, 45, 46, 52, 54, 60] aiming to map relational databases (RDB) to ontologies. In existing methods for ontology engineering from a relational database, a mapping is defined aiming to transform database elements to ontology elements. After an analysis of these articles, nineteen mapping rules used to map relational database elements to ontology elements were identified. Rules are presented in Table 2.1. Following, we present the related works and the details of proposed mapping rules.

Table 2.1: Common rules of mapping relational database to ontology

Mapped Elements	Rule	Rule Description
1. Tables to classes	A	It maps each non-associative table to a class.
	B	It integrates information from different tables in a single class.
2. Associative tables to object properties	C	It maps many-to-many relationships to object property.
	D	It maps many-to-many relationships to two mutually inverse object properties.
	E	It defines domain and range for object properties which represent many-to-many relationships.
	F	It maps each table which references more than two tables to one class and one object property for each FK.
3. Foreign key (FK) to object properties	G	It maps FKs in non-associative tables to object property.
	H	It defines domain and range for object properties which represent FK in rule G and I.
	I	It maps FK to two object property mutually inverse.
	J	It defines “has-part” “and is-part-of” object properties when FK is part of the primary key in non-associative tables.
4. Tables to subclasses	K	It maps a subclass for FK which is equivalent to the PK.
	L	It maps a subclass for each FK which is part of the PK.
5. Columns to data properties	M	It creates one datatype property for each attribute which cannot be mapped to an object property.
6. Primary key (PK) constraints to properties	N	It defines min and max cardinalities equals to 1 for properties which represent PK attributes.
	O	It maps PK attributes with min cardinality equals to 1 and as inverse functional properties.
7. Unique (UK) constraints to properties	P	It defines max cardinality equals to 1 for data properties which represent UK attributes.
	Q	Map UK columns as functional properties.
8. Not null constraints	R	It maps not null attributes with min cardinality restriction equals to 1.
9. Table data to instances	S	It maps table data to instances.

1. **Tables to classes:** each table which is not an associative table (e.g. those which does not represent pure many-to-many relationships) is mapped to an ontological class in rule A [9, 13, 16, 25, 34, 36, 45, 46, 52, 60]; additionally, when tables have

¹Tissot, Hegler Correa; Del Fabro, Marcos Didonet. Mapping relational databases to ontologies. Informatics Department, Federal University of Paraná, Brazil, 2014. Not published.

the same PK definition [52], or several tables are used to describe one entity [34], such information is integrated in one single ontological class in rule B [34, 52].

2. **Associative tables to object properties:** each associative table which represents a many-to-many relationship between two other tables is mapped to two object properties in rule C [9, 45, 46, 52], which can be also defined as object properties mutually inverse in rule D [16, 34, 36, 60]; domain and range are defined for such object properties in rule E [16, 45, 46, 52, 60]; however, for tables which reference more than two tables (multi-relationships), a class is created to represent the bridge table, and one object property is defined for each FK in rule F [34, 36].
3. **Foreign key (FK) to object properties:** for tables which are not associative, each FK is mapped to an object property in rule G [9, 16, 25, 34, 45, 52, 60]; domain and range are defined for such object properties in rule H [16, 25, 45, 52, 60]; moreover, [36, 52] use an approach to map each FK to two object properties in rule I: a) the first object property has as domain the class corresponding to table containing the column, and its range is the related table of the FK column, and b) second object property is declared as inverse functional of the first FK object property [36, 52]; for each entity described by a relation which is a special type of entity described by other, i.e. FK which connects both tables is also part of the PK is defined two object properties named “has part” and “is part” in rule J [34, 52, 60].
4. **Tables to subclasses:** different solutions consider specific mapping rules to define subclasses in target ontology, including related tables which share same PK definition, i.e. FK is equivalent to PK in rule K [9, 36, 45, 52], and related tables which have a FK as part of PK in rule L [16, 34, 46, 60].
5. **Columns to data properties:** one datatype property is created for each column which cannot be mapped to an object property in rule M [9, 13, 16, 25, 34, 36, 45, 46, 52, 60]; in [46], however, primary key attributes are not defined as data properties even when they do not represent FK attributes.

6. **Primary Key (PK) constraints to properties:** while some approaches define min and max cardinalities equals to 1 for data properties, which represent PK attributes in rule N [13, 34, 52], others define min cardinality of PK attributes equals to 1 and define such attributes as inverse functional properties, aiming to ensure uniqueness in rule O [9, 36, 45].
7. **Unique (UK) constraints to properties:** similar to the approach used to map primary key, max cardinality is defined equals to 1 for data properties which represent Unique key (UK) attributes in rule P [34, 52], and others define such attributes as inverse functional properties in rule Q [9, 36, 46].
8. **Not null constraints:** min cardinality restriction is defined equals to 1 for ontological properties which represent not null attributes in rule R [9, 34, 36, 46, 52].
9. **Table data to instances:** each tuple is mapped to one ontological instance of its specific class, following the definition of data and object properties in rule S [9, 25, 32, 34, 36, 45, 46, 52, 54, 60].

Related works present, in short, mapping rules which map RDB tables, attributes and data to the corresponding ontology components of classes, properties, and instances.

Astrova [9] has made a considerable contribution to this research area. In [10], the author explain rules of mapping databases to ontologies, considering mapping rules to database tables, columns, and tuples. In [9], it added check constraint mapping rule. An approach to mapping data types from SQL to XSD was presented by [10] and [13].

Moreover, [45] propose the creation of cardinality property restriction, intend to map constraints at column level from data type information. They also proposed semantic rules which defines *equivalence class*, *all values from* and *some values* property definitions. The main intuition of these rules was to promote the ontology-based RDB development with more reasoning support.

In [46], authors propose a framework which classifies data according to metamodels and map constraints in relational databases. In the metamodel, data are divided into two parts: master data, which represents core objects in business; and transactional data,

which represents transactional flow tuples in business behaviors. Based on this classification, the proposed framework performs a transformation process using the schema and the data level, to generate two ontologies. Each ontology is generated for each transformation process.

Zhang and Li [60] propose an automatic method to map a relational database to ontologies. The methodology was developed using metamodels concepts. They extract data from a relational database to a database metadata model. After its first mapping, its metadata is mapped to an ontology meta-model by mapping rules. Finally, OWL document is generated using the ontology meta-model. As a conclusion, they evaluated a huge contrast between the ontology produced automatically and the one created manually; the automatic generation requires further research.

Two distinct tools DB2OWL [16] and RDB2OWL [13] automatically generate ontologies from database schema. DB2OWL [16] generates, from mapping process, a document with relations of database and ontologies elements and RDB2OWL [13] elaborates a mapping schema to establish the correspondences between RDB schema elements and OWL ontology elements.

Telnarova [52] focused on the principles of automatic conversion of ontology elements, mapping relational data model to OWL ontology elements and data to ontology instances. Louhdi [36] proposed rules which analyze stored data to detect disjointness and totalness constraints in hierarchies.

Two other solutions [32, 54] propose to map exclusively database tuples to ontological instances. In [32] the authors extract tuples from a schema and map it using a specific template. In [54] authors execute queries which extract tuples from RDB to generate ontology tags and their solution was presented with a friendly interface.

Regarding mapping check constraints, [45] proposed a different mapping that led us to set a new rule for mapping check constraints. In [46], the authors propose naming the properties by concatenating the name of the class to the property name, seeking not to generate duplicated elements. However, we noticed that the solution of [46] can generate countless properties containing the same represented concept, differing only the

domain and range of the property. This made us reflect upon the need a mapping that eliminates duplicated elements, not only for property elements but also for classes and instances. Another contribution made in [46] lies in the table data classification. We use this specification to classify different kinds of transformations according to the table data content.

In general, related works do not present details about naming ontological elements during the mapping process. An exception is [46] which append the database relation name to the column name when mapping columns. In related works, the name of database elements comes from database schema and the name of ontology elements retain the source name of database element. When there is a type of encoding in the name of database elements, is more difficult to understand the meaning of target ontology. The encoding is commonly declared to arrange the database elements in the source RDB schema.

Different types of mapping have been proposed to establish a relation between the concepts of database elements and the concepts of ontology elements, in order to construct an ontology from a database. Some proposals are not clear on how the Mapping Process is performed [52, 34]. Others cite that the process is performed semi-automatically or automatically [16, 45, 46, 54, 60]. In this work we propose to develop an architecture that unifies the rules A, D, E, F, G, H, I, K, L, M, O, Q, and S of related works presented in Table 2.1 and we propose new rules for the mapping process.

CHAPTER 3

MAPPING ARCHITECTURE

Relational database mapping to ontologies is a subject explored by many researchers, as we presented in 2.4. As for the main objectives, they intend to use database structure aiming: simplify the build of the ontology; allow multiple database integrations; learning of ontology; managing knowledge of an organization using ontology and others [17].

As principles of the elaborated mapping architecture, we seek an increase in the legibility of the generated ontology and the elimination of duplicated elements, due to the defined criteria for naming them and the mapping structure which allows database elements to be related to one or many elements of the ontology.

Our proposed mapping architecture in this work is composed of external and internal components. As external components, we have the Database Schema and Logical Model, used as input artifacts to fill the Configuration Template. As internal components we have: a) the Configuration Template, which provides database source information; B) the Mapping Model, a metadata repository where the database elements are mapped to ontology elements; C) the Mapping Process, which receives the contents of Configuration Template to perform the Mapping Process and store the results in a schema based on Mapping Model; and d) the Generation of Ontology, which from the results stored in a schema based on Mapping Model, generates an ontology. With the definition of the Mapping Model, Mapping Process and Generation of Ontology we have developed a Prototype to integrate the execution of these components.

We present in this chapter our proposed work. Section 3.1 presents in detail each component of our mapping architecture. Section 3.6 presents the Prototype specification to perform the Mapping Process from relational database (RDB) to the generation of ontology.

3.1 Architecture components

The proposed mapping architecture of this work was designed based on proposals suggested by various previous works as well as from observed needs. Through the analyzes the proposal of several authors we developed a mapping architecture that allows storing the mapping between database elements and ontology elements, offering a more complete solution than the one proposed by [16, 13]. For mapping rules we rewrite 13 rules (A, D, E, F, G, H, I, K, L, M, O, Q, and S) of related works presented in Table 2.1 and we considered 3 rules proposed by Tissot ¹. Our contribution consists in the definition of 3 new rules and in the adequacy of rules and proposals, aiming to eliminate duplicate elements, however, maintaining the traceability between the database and ontology elements.

We also consider in our mapping architecture the Logic Model rather than using uniquely database schema or physical model. The terminology used in the Logic Model are more related to the real world definitions than the terminology used in the physical model, improving the semantics of the generated ontology elements.

From the presented considerations we have elaborated a mapping architecture capable of overcoming these needs. Figure 3.1 presents the Configuration Template and External and Internal Components that constitute the architecture. External Components have Database Schema and Logic Model as subcomponents. Data of External Components is filled into Configuration Template. Internal Components are composed of subcomponents Mapping Process, Mapping Model and Generation of Ontology. Owl Ontology is a subcomponent generated from the others Internal subcomponents. For a better understanding of architecture components, we present each component in detail in the coming sessions.

¹Tissot, Hegler Correa; Del Fabro, Marcos Didonet. Mapping relational databases to ontologies. Informatics Department, Federal University of Paraná, Brazil, 2014. Not published.

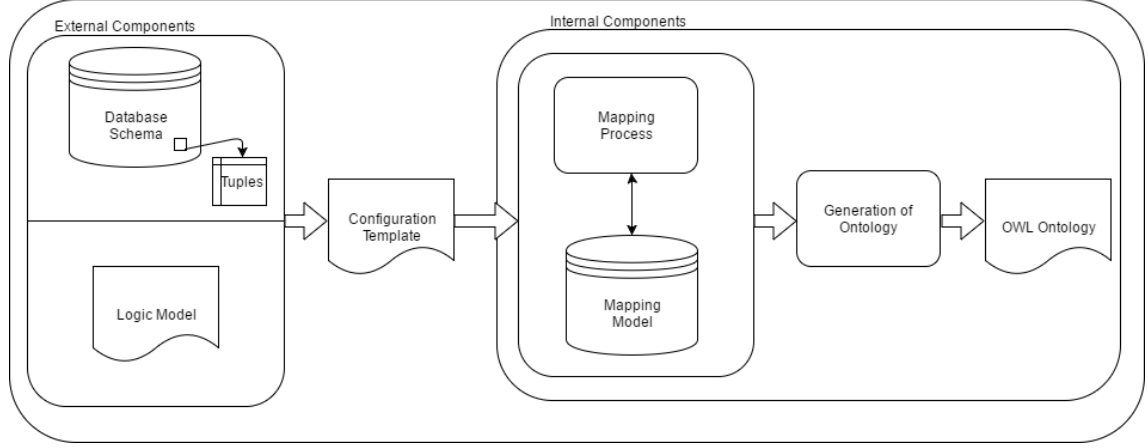


Figure 3.1: Architecture components

3.1.1 Configuration template

In order to import the database structure and considering the use of the Database Schema, Tuples, and Logical Model, a template was set so that all this information could be inserted into a single file and so that a prototype was capable of reading it at once. Initially, we considered creating an automatic process, from the start of the architecture to its end. As we studied mechanisms that enable the extraction of the Database Schema, Tuples, and Logical Model, we perceived that recent studies directed efforts in this sense [12] and that the complexity involved in this activity could be higher than the complete conception of the proposed architecture.

Based on this and on a large amount of database management systems to be considered in an integration, we designed a template in Comma Separated Value (CSV) format, commonly used in the area [40]. The Configuration Template was developed to receive the data originated from a) the Database Schema; b) the Logic Model; c) Tuples; and d) the indication of specific types, such as the range of values of a check constraint, concept definitions of a check constraint, classification of fields of the description type and classification of table types.

A relevant factor in the definition of the structure of this template is the possibility to partially import the database structure, not making it mandatory to inform the database tuples or Logical Model. Another relevant point in the proposed architecture is that with

the definition of Configuration Template we do not restrict the use of the architecture to a specific database or a tool, in case of using the Logic Model.

The file is filled manually, which allows the input of the data to be previously filtered when the extraction of tuples is done or it can even be increased when the concept that represents the range of values of a constraint is filled in. Appendix A presents definitions of each item which composes the Configuration Template.

3.1.2 Mapping model

During an analysis of the use of ontology in data integration, Cruz [15] affirmed the possibility of use a standard ontology to support the mapping from a database to ontologies. The term “mapping” is often used to refer to the ontology connection to the other parts of application systems [57]. In order to allow the mapping between relational database elements and ontology elements, we designed a structure to perform this correspondence. For this, we used the ontology concept for modeling this structure as a mediator conceptual schema [37].

According to Ushold [53], the RDB schema can serve as ontological structure, where it is possible to specify relations and integrity constraints. Based on this statement, we created a mapping to represent the concepts and the relations in entity–relationship model (ER model). We represented the elements from database and ontology, the relation of the elements, and its correspondences by tables relationship, respecting the definitions of mapping rules.

The Mapping Model is divided into two parts: part A, in Figure 3.2, represents concepts related to database and part B, in Figure 3.3, represents concepts related to ontology. The area comprising the concepts related to relational database comprises the tables: T001_database, T002_table, T003_column, T004_record, T005_datatype_db, T006_check_value, T007_check_subject, T008_database_domain, T009_column_check_value, T010_table_db_domain.

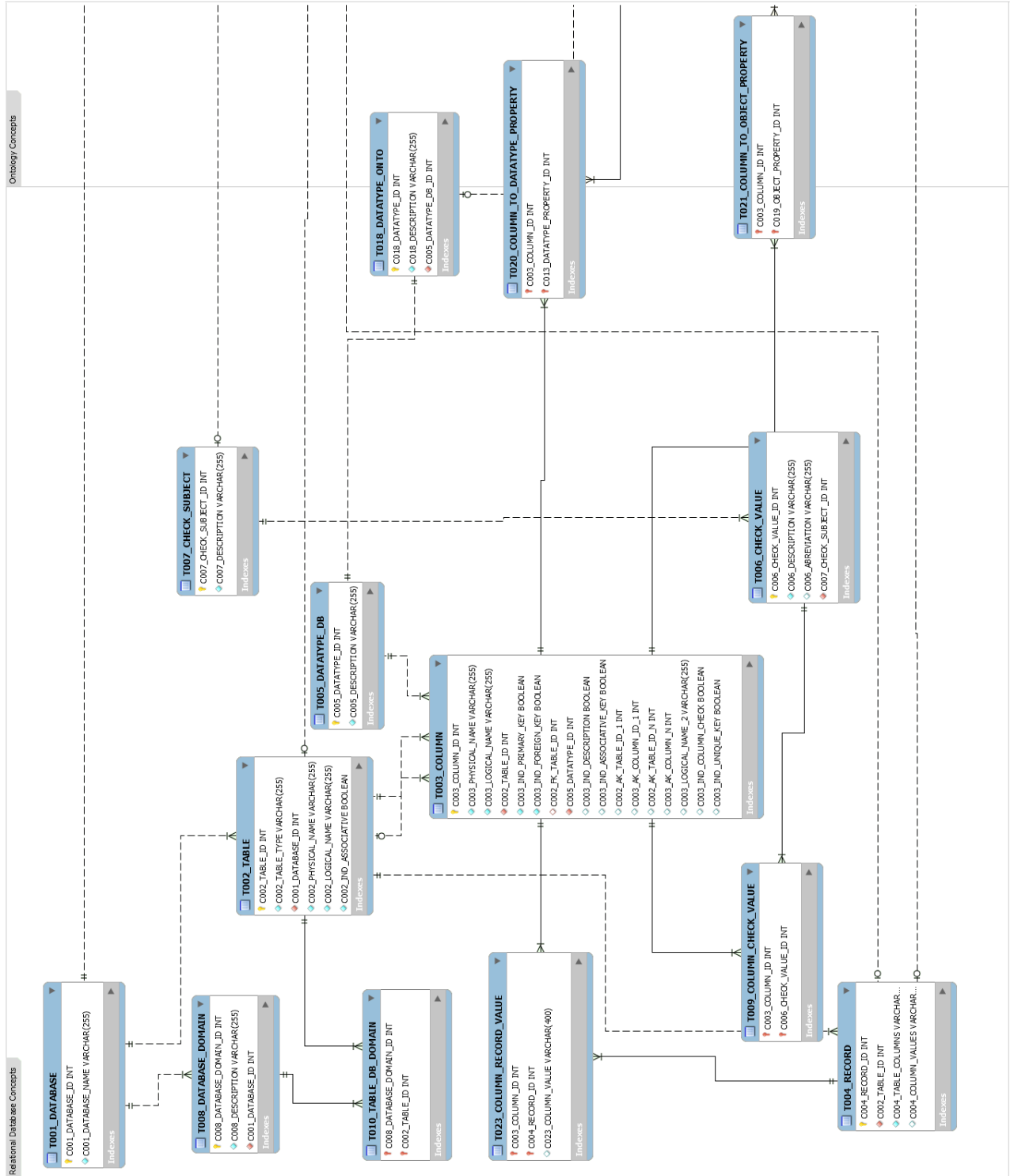


Figure 3.2: ER model - part A - mapping of database elements

The area comprising the concepts related to ontology contain the tables: T011_class, T012_hierarchy, T013_datatype-property, T014_datatype-property-domain, T015_instance, T016_ontology, T017_disjoint_class, T018_datatype-onto, T019_object-property, T022_object-property-domain-range, T023_column-record-value.

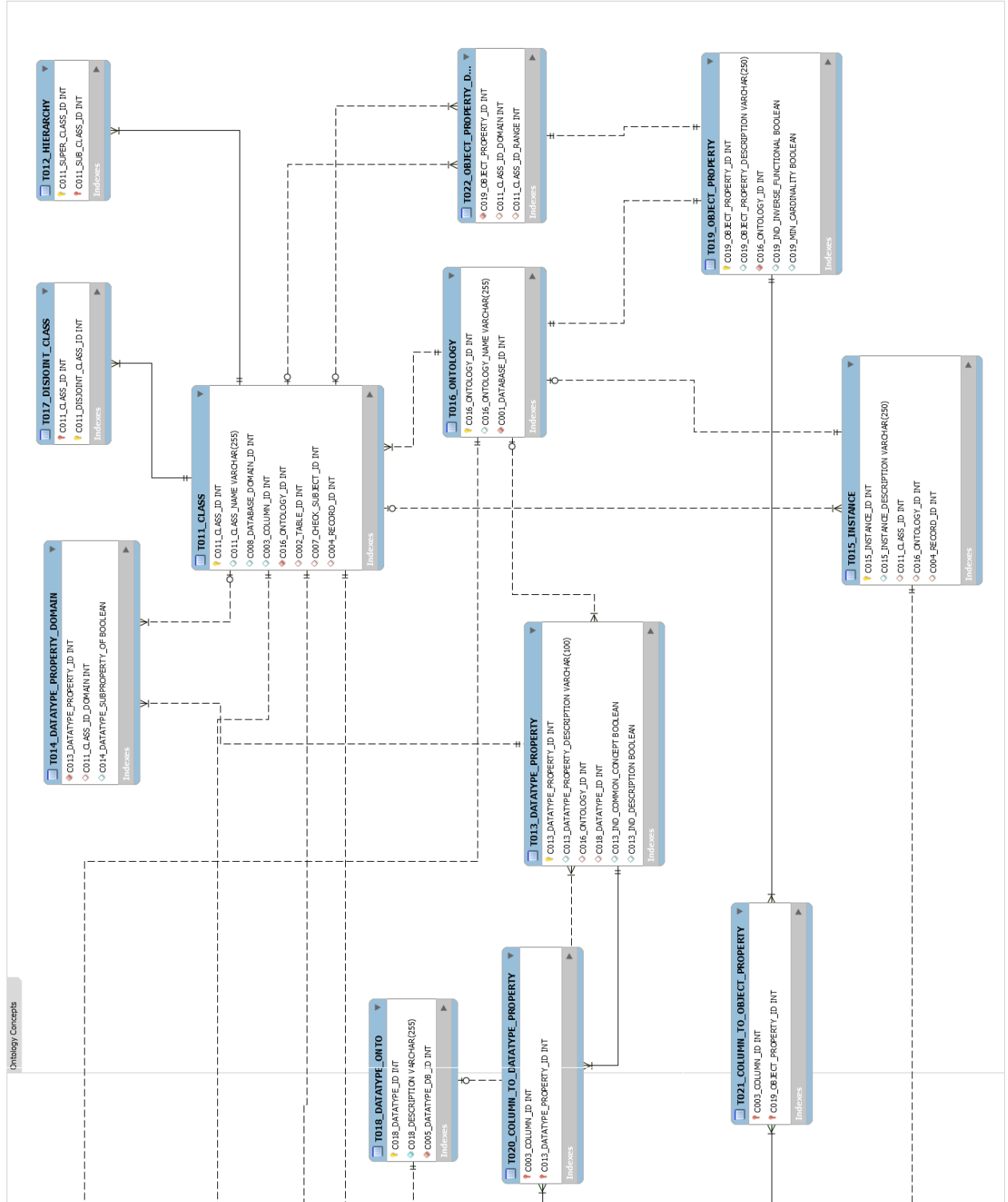


Figure 3.3: ER model - part B - mapping of ontology elements

Tables T020.column.to.datatype.property and T021.column.to.object.property were defined to relate database and ontology concepts. The complete Mapping Model can be seen in Appendix B, and Appendix C presents details of each table of Mapping Model.

3.1.3 Mapping process

In our architecture, the mapping process is performed by a set of rules. Each rule contains a definition of the relations to be established between database elements to ontology elements, definitions regarding the naming of elements and eventual definitions of exceptions or restrictions to be considered.

In the Mapping Process that we defined, we present details of how we perform the naming of the elements, how we correlate SQL data types to XML data types, and the rule definitions of the Mapping Process, for the latter we rewrote proposed rules in previous works and we also propose new rules. Figure 3.4 shows the Mapping Process component, which receives the information to be mapped by the Configuration Template and as it performs the mapping. During the Mapping Process, an instance of the schema generated from the Mapping Model stores the database and ontology elements and also mapping the relations among them.

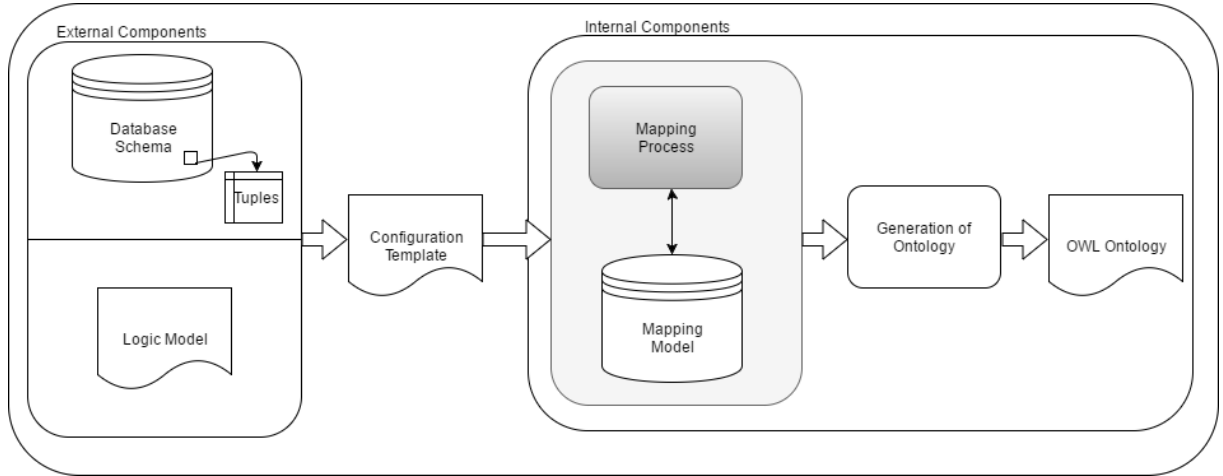


Figure 3.4: Components of mapping process

3.1.3.1 Naming ontology elements

Conventions for naming classes and properties are proposed in [31]. Although there are no mandatory naming conventions for OWL classes, it is recommended that every class name should start with a capital letter and should not contain spaces. With regard to properties, there is also no naming convention. Property names should start with a lower

case letter, no spaces, and the remaining words capitalized. It is also recommended to prefix properties with the word “has”, or the word “is”, for example *hasPart*, *isPartOf*, *hasManufacturer*, *isProducerOf*. Proposed conventions by [31] help on property identification and its understanding, and also establish a pattern of ontology reading.

When we consider only the names of elements from the database schema, we depart from the essence of building a common vocabulary which provides a common understanding of knowledge domain. We observed in the elements name of physical model the use of numbers and characters, as underscore and underline, to organize table structures, which is commonly used to separate sentences and words. When, e.g. *T009_Proc_Covered_Plan* table is converted to ontological class, and the class is named to *T009_Proc_Covered_Plan* the mapped ontology element is difficult to be decoded, whether mapping rules consider only the physical name of elements without criteria to format it.

Thus, in our architecture, we chose to use the information from the Logic Model for naming ontology elements. In the absence of information from the Logic Model, we consider the information from the Database Schema. Front of this and to improve semantic characteristics from database elements to ontology elements, we additionally perform a filtration in the element names to remove characters, by defining in our Prototype a formatting function. The filtration is applied during naming the ontology elements, in the Mapping process.

Formatting function is used to the formatting of the physical and logical name of database elements and when the logical name is filled in, the physical name is discarded from the Mapping Process. One of the treatments of our formatting function treats each space between the words, removing them. Formatting function also removes words with less than three characters, e.g. (*in*, *on*, *at*, *of*, *by*) underscore and underline characters. When the character is removed, the first letter of each word is highlighted in capital letter. With the formatting function *T009_Proc_Covered_Plan* table is converted to ontological class named as *ProcCoveredPlan*.

3.1.3.2 Data types

As proposed by Astrova [10], we correlate SQL data types from database elements to XML data types to ontology elements. For this purpose, the data types of the SQL Server, Oracle, MySQL and PostgreSQL databases were considered. The size constraint specifications defined in the data types of database columns were disregarded. We do not consider SQL data types such as blobs (binary large objects), image and xml in the list of correlation between SQL and XML data types. For cases where the data type is not in the list of correlated elements, we used the classification of the XML data type *string*. Table 3.1 displays the SQL data types match definitions for XML data types.

Table 3.1: Correlated data types

SQL data types (Sql Server, Oracle, MySQL, PostgreSQL)	XML data types
Text, TinyText, MediumText, LongText, Varchar2, varchar, char, nchar, nvarchar, nvarchar2, lob, and long	string
Integer, Int, Mediumint, Numeric, smallint, tinyint, bigint, and decimal	decimal
Bit, Byte, and Boolean	boolean
Float, Real, Number, and Numeric	float
Double	double
Date, year	date
Time	time
Timestamp	datetime
Interval	duration

3.1.3.3 Mapping rules

We present in the following subsections each rule and how the Mapping Process is performed. We describe how we can identify the information to be mapped in the Configuration Template; we also describe which database elements are mapped to which ontology elements and how these elements are named; for a better understanding of definitions of the proposed rules, we present a pseudocode of each rule; and, we present an example of generated OWL tags from rule following W3C standards is presented. The rules respect the definitions presented in Subsections 3.1.3.1 and 3.1.3.2.

Mapping tables

Rule 1: mapping non-associative tables

Non-associative tables store primary information to be used as reference to other tables. We can identify non-associative tables when a table contains only one primary key or whether the table contains primary key columns that are not foreign keys of two other distinct tables.

In the Configuration Template we defined the field *Type* to identify tables (T), columns (C) and tuples (R), as we presented in Subsection 3.1.1. For each table (T), the field *Is_Associative* in the Configuration Template can have two table classifications: (1) for associative tables or (0) for non-associative tables. Each non-associative table is mapped to one ontological class, wherein the class name is composed of: a) it starts with a capital letter and b) the value which correspond to the table name. This rule is equivalent to Rule A of Table 2.1. The sequence of steps of this mapping rule is displayed in Algorithm 3.1.

Algorithm 3.1: Rule 1 - mapping non-associative tables

```

1 read configurationTemplate;
2 tables ← configurationTemplate.type(T);
3 insert into MappingModel.T002_TABLE(tables);
4 for (i = 0; tables is not null; i++) do
5     table ← tables[i];
6     if table is non-associative then
7         class ← formattingFunction(table);
8         insert table, class into MappingModel.T011_CLASS;
9     else
10        go to Algorithm 3.2;
11    end
12 end
13 Result: Mapping non-associative tables to classes.

```

Considering the table *T002_Dentist* are non-associative, it is mapped to the class

Dentist represented by the following OWL tags in the generated ontology:

Example 9:

```
<Declaration>
    <Class IRI="#Dentist"/>
</Declaration>
```

Rule 2: mapping associative tables

Associative tables represent a relationship between two entities. We can identify associative tables when their primary key columns are only foreign keys to two other tables. In rule 1 we explained how to identify associative tables in the Configuration Template. For mapping associative tables we also defined the field *Is_Primary_Key* to identified primary key columns(1), and the field *Related_Table*, in which is filled the referenced table of the foreign key columns.

At first, we map the primary key columns of associative tables to ontological elements generating an object properties (mutually inverse) for each primary key column, unless the table columns are only foreign key. The first object property name is composed of: a) it starts with fixed value *has* - with lowercase letter; and b) the corresponding value for the column name - starts with a capital letter. The second object property name is composed of: a) it starts with fixed value *is* - with lowercase letter; b) the corresponding value for the column name - starts with a capital letter; and c) the fixed value *of* - with a capital letter.

For each object property a minimum cardinality of restriction, declared as inverse, and the *domain* and *range* are assigned. The *range* is specified using the created class to the foreign key of the referenced table; and *domain* is specified using the created class to the referenced table to the other foreign key which compose the associative table. Algorithm 3.2 contains the pseudocode of this mapping rule.

Algorithm 3.2: Rule 2 - mapping associative tables

```

1  read configurationTemplate;
2  columns  $\leftarrow$  configurationTemplate.type(C);
3  insert into MappingModel.T003.COLUMN(columns);
4  insert into MappingModel.T005.DATATYPE_DB(columns.dataType);
5  for ( $i = 0$ ; tables is not null;  $i++$ ) do
6      table  $\leftarrow$  tables[i];
7      if table is associative then
8          for ( $j = 0$ ; table.columns is not null;  $j++$ ) do
9              column  $\leftarrow$  table.columns[j];
10             if column is primary key then
11                 1ObjProp  $\leftarrow$  “has”+formattingFunction(column);
12                 2ObjProp  $\leftarrow$  “is”+formattingFunction(column) + “of”;
13                 1ObjProp, 2ObjProp  $\leftarrow$  isMinCardinality, isInverse, column.domain,
14                     column.range;
15                 insert column, 1ObjProp, 2ObjProp into
16                     MappingModel.T021.COLUMN_TO_OBJECT_PROPERTY,
17                     MappingModel.T019.OBJECT_PROPERTY and
18                     MappingModel.T022.OBJECT_PROPERTY_DOMAIN_RANGE;
19             else
20                 go to Algorithm 3.3;
21             end
22         end
23     else
24         end
25 end
26 Result: Mapping associative tables to object properties.
  
```

In Example 10, the associative table *T024.Dentist_Specialty* is mapped to the object properties (mutually inverse) named *hasDentistId*, *isDentistOf*; and *hasSpecialtyId*, *isSpecialtyIdOf*, according to their primary key columns. The *hasDentistId* object property is represented by the following OWL tags in the generated ontology:

Example 10:

```

<Declaration>
  <ObjectProperty IRI="#hasDentistId"/>
</Declaration>

<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasDentistId"/>
  <Class IRI="#Especialidade"/>
</ObjectPropertyDomain>

<ObjectPropertyRange>
  <ObjectProperty IRI="#hasDentistId"/>
  <ObjectMinCardinality cardinality="1">
    <ObjectProperty IRI="#hasDentistId"/>
    <Class IRI="#Dentista"/>
  </ObjectMinCardinality>
</ObjectPropertyRange>

<InverseObjectProperties>
  <ObjectProperty IRI="#hasDentistId"/>
  <ObjectProperty IRI="#isCodigoDentistaOf"/>
</InverseObjectProperties>

```

We can identify in the schema generated from the Mapping Model whether an associative table has additional columns, finding a column which is not checked as primary key. Whether the result returns a value we apply in conjunction with the rule above, these steps: a) we create a class to the associative table, according to Rule 1; and b) we create a datatype property for each non primary key column, according to Rule 3. This rule is equivalent to Rules D, E and partly to F of Table 2.1. The pseudocode of this extra mapping is described in Algorithm 3.3.

Algorithm 3.3: Rule 2 - mapping associative tables with non primary key columns

```

1 continue from line 10 of Algorithm 3.2;
2 if column is not primary key then
3   | Algorithm 3.2;
4 else
5   | class ← formattingFunction(table);
6   | insert table, class into MappingModel.T011_CLASS;
7   | datatypeProp ← formattingFunction(column), column.domain, column.dataType;
8   | insert column, datatypeProp into
      | MappingModel.T020_COLUMN_TO_DATATYPE_PROPERTY,
      | MappingModel.T013_DATATYPE_PROPERTY,
      | MappingModel.T014_DATATYPE_PROPERTY_DOMAIN and
      | MappingModel.T018_DATATYPE_ONTO;
9 end
10 Result: Mapping associative tables to class and datatype property.

```

Mapping columns

Database represents characterizes an information type or category to be included in the table. For each table column defined in the source relational database, we define a datatype property or an object property. In the Configuration Template we can identify the different column types by the fields: primary key (*Is_Primary_Key*), unique key (*Is_Unique_Key*) foreign key (*Is_Foreign_Key*) or check constraint (*Is_Column_Check*).

Rule 3: mapping columns to datatype properties

Table columns that are not part of these column types (primary, unique and foreign key or check constraint) are mapped to a datatype property, at first moment, wherein the datatype property name is composed of: a) it starts with lowercase letter; and b) the value corresponding to the column name.

For each datatype property the *domain* and *range* are assigned, where the *domain* is specified using the created class to the table column; and *range* is specified using the

column data type value. Each datatype property is related to the respective *domain* and *range*. The definition above is equivalent to Rule M of Table 2.1. Algorithm 3.4 displays the pseudocode of this mapping rule.

Algorithm 3.4: Rule 3 - mapping columns

```

1  continue from line 7 of Algorithm 3.2;
2  for ( $i = 0$ ; table.columns is not null;  $i++$ ) do
3      column  $\leftarrow$  table.columns[i];
4      if column is not (primary key, unique key, foreign key, check constraint and
        description) then
5          datatypeProp  $\leftarrow$  formattingFunction(column), column.domain, column.dataType;
6          insert column, datatypeProp into
              MappingModel.T020_COLUMN_TO_DATATYPE_PROPERTY,
              MappingModel.T013_DATATYPE_PROPERTY,
              MappingModel.T014_DATATYPE_PROPERTY_DOMAIN and
              MappingModel.T018_DATATYPE_ONTO;
7      else
8          if column is description then
9              go to Algorithm 3.5;
10         else
11             if column is primary key or is unique key or is foreign key then
12                 go to Algorithm 3.6;
13             else
14                 go to Algorithm 3.9 and Algorithm 3.10;
15             end
16         end
17     end
18 end
19 Result: Mapping no constraint columns to datatype properties.

```

As we present in Example 11, the *T002_Dentist* table has *A006_Dentist_Name* column, which is mapped to a datatype property named *Dentistname*. After the mapping, the elements are represented by the following OWL tags in the generated ontology:

Example 11:

```

<Declaration>
  <DataProperty IRI="#Dentistname"/>
</Declaration>

<DataPropertyDomain>
  <DataProperty IRI="#Dentistname"/>
  <Class IRI="#Dentist"/>
</DataPropertyDomain>

<DataPropertyRange>
  <DataProperty IRI="#Dentistname"/>
  <Datatype abbreviated IRI="xsd:string"/>
</DataPropertyRange>

```

Ontological elements may have conflict when setting datatype property names for columns that have the same physical or logical names in different tables (e.g. columns with physical name *active* and columns with logical name *description*). In this case, only one datatype property should be created to refer to all database columns which have the same logical name.

In the Configuration Template we can identify columns which describes items from a specific domain by the field *Is_Description*. This condition was parameterized in the template because the tables can use different names to identify a column type which contains description of the represented context. We can create only one datatype property for all columns checked as description in the Configuration Template, considering that the meaning of this property is the same, only change the data in each table. For this columns, the datatype property name is composed of: a) it starts with lowercase letter; and b) the fixed value *description*.

Through one datatype property is created to refer all database columns which have the same logical name, datatype property domain has to be defined considering all tables which have columns description. Therefore, to the description datatype property the *domain* and *range* are assigned, where the *domain* is specified using the created class for each description column; and *range* is specified using the column data type value.

For each datatype property one subproperty, the *domain* and *range* are assigned. The subproperty connect in a hierarchical structure to the first one *description* datatype property; *domain* is specified using the created class to the table column; *range* is specified using the column data type value. After the mapping, the elements are represented by the following OWL tags in the generated ontology:

Example 12:

```

<Declaration>
  <DataProperty IRI="#description"/>
</Declaration>

<DataPropertyDomain>
  <DataProperty IRI="#description"/>
  <ObjectUnionOf>
    <Class IRI="#Procedure"/>
    <Class IRI="#UserType"/>
    <Class IRI="#MaritalStatus"/>
  </ObjectUnionOf>
</DataPropertyDomain>

<DataPropertyRange>
  <DataProperty IRI="#description"/>
  <Datatype abbreviated IRI="xsd:string"/>
</DataPropertyRange>

```

Additionally, for each field checked as description in the Configuration Template, a datatype property is mapped. Its name is composed of: a) it starts with the class name mapped from the table column - with lowercase letter; and b) the fixed value *Description* - with a capital letter. The sequence of steps of this mapping rule is displayed in Algorithm 3.5.

Algorithm 3.5: Rule 3 - mapping description columns

```

1 continue from line 10 of Algorithm 3.4;
2 datatypePropDesc ← formattingFunction(“description”);
3 descriptionColumns ← listAlldescriptionColumns();
4 for ( $i = 0$ ; descriptionColumns is not null;  $i++$ ) do
5   column ← descriptionColumns[i];
6   datatypePropDesc ← column.domain, column.dataType;
7   datatypeProp ← formattingFunction(column), column.domain, column.dataType;
8   datatypeProp is subproperty of datatypePropDesc;
9   insert column, datatypePropDesc, datatypeProp into
      MappingModel.T020_COLUMN_TO_DATATYPE_PROPERTY,
      MappingModel.T013_DATATYPE_PROPERTY,
      MappingModel.T014_DATATYPE_PROPERTY_DOMAIN and
      MappingModel.T018_DATATYPE_ONTO;
10 end
11 Result: Mapping description columns to datatype properties.

```

This elements are represented by the following OWL tags in the generated ontology:

Example 13:

```

<Declaration>
  <DataProperty IRI=“#procedureDescription”/>
</Declaration>

<DataPropertyDomain>
  <DataProperty IRI=“#procedureDescription”/>
  <Class IRI=“#Procedure”/>
</DataPropertyDomain>

<SubDataPropertyOf>
  <DataProperty IRI=“#procedureDescription”/>
  <DataProperty IRI=“#description”/>
</SubDataPropertyOf>

<DataPropertyRange>
  <DataProperty IRI=“#procedureDescription”/>
  <Datatype abbreviated IRI=“xsd:string”/>
</DataPropertyRange>

```

In addition, regardless of whether the description field has been filled in or not, we verify columns have the same physical or logical name in different tables, to avoid conflicts. In this case only one datatype property for all columns with repeated name is created. For this datatype property the *domain* and *range* are assigned, where the *domain* is specified using the created class for each column with duplicated name.

Rule 4: mapping columns to object properties

In the Configuration Template, the fields checked as primary or unique key are mapped to two functional object properties (mutually inverse), wherein the first object property name is composed of: a) it starts with fixed value *has* - with lowercase letter; and b) the value corresponding to the column name - starts with a capital letter. The second object property name is composed of: a) it starts with fixed value *is* - with lowercase letter; b) the value corresponding to the column name - starts with a capital letter; and c) the fixed value *of* - with a capital letter.

Each object property is declared as functional, inverse and its *range* is specified using the created class from the table column. Object property mapped from primary key a minimum cardinality of restriction is assigned. The definition above was partially proposed in Rules O and Q of Table 2.1.

The *T008_Procedure* table has the column *A008_Procedure_Id* checked as its primary key and the column *A008_Procedure_Number* checked as unique key. Each column is mapped to two object properties, wherein object properties generated from the mapped primary key are named as: *hasProcedureId* and *isProcedureIdOf*; and the unique key mapped to object properties are named as *hasProcedureNumber* and *isProcedureNumberOf*.

As we map the primary and unique key to object properties, for each object property we define the *range*, which this *range* is defined to *Procedure* Class. After performing the mapping, as we present in Example 14, the elements are represented by the following OWL tags in the generated ontology:

Example 14:

```

<Declaration>
  <ObjectProperty IRI= "#hasProcedureId"/>
</Declaration>

<Declaration>
  <ObjectProperty IRI= "#isProcedureIdOf"/>
</Declaration>

<InverseObjectProperties>
  <ObjectProperty IRI= "#hasProcedureId"/>
  <ObjectProperty IRI= "#isProcedureIdOf"/>
</InverseObjectProperties>

<FunctionalObjectProperty>
  <ObjectProperty IRI= "#hasProcedureId"/>
  <ObjectProperty IRI= "#isProcedureIdOf"/>
</FunctionalObjectProperty>

<ObjectPropertyRange>
  <ObjectProperty IRI= "#hasProcedureId"/>
  <ObjectMinCardinality cardinality= "1">
    <ObjectProperty IRI= "#hasProcedureId"/>
    <Class IRI= "#Procedure"/>
  </ObjectMinCardinality>
</ObjectPropertyRange>

<ObjectPropertyRange>
  <ObjectProperty IRI= "#isProcedureIdOf"/>
  <ObjectMinCardinality cardinality= "1">
    <ObjectProperty IRI= "#isProcedureIdOf"/>
    <Class IRI= "#Procedure"/>
  </ObjectMinCardinality>
</ObjectPropertyRange>

```

In the Configuration Template, the fields checked as foreign key (in non-associative tables) is mapped to two object properties (mutually inverse). Their names respect the primary and unique key definition presented above.

For each object property minimum cardinality of restriction, declared as inverse, and the *domain* and *range* are assigned. The *range* is specified using the class created from the foreign key of the referenced table; and *domain* is specified using the class corresponding

the referenced table of the foreign key column. The definition for foreign key is equivalent to Rules G, H and I of Table 2.1 and we adding new features. Algorithm 3.6 contains the pseudocode of this mapping rule.

Algorithm 3.6: Rule 4 - mapping columns

```

1 continue from line 13 of Algorithm 3.4;
2 objectProp  $\leftarrow$  formattingFunction(column);
3 if column is primary key then
4   | objectProp  $\leftarrow$  column.range;
5   | objectProp is functional, inverse and has minCardinality;
6 else
7   | if column is unique key then
8     | objectProp  $\leftarrow$  column.range;
9     | objectProp is functional and inverse;
10  | else
11    | objectProp  $\leftarrow$  column.domain, column.range;
12    | objectProp is inverse and has minCardinality;
13  | end
14 end
15 insert column, objectProp into
    MappingModel.T021_COLUMN_TO_OBJECT_PROPERTY,
    MappingModel.T019_OBJECT_PROPERTY and
    MappingModel.T022_OBJECT_PROPERTY_DOMAIN_RANGE;
16 Result: Mapping columns to object properties.

```

Mapping tuples

Database tuples is an ordered list of n values where each value is an element of domain, consisting in a set of data structured in a table. We defined three categories to classify tables according to their content: specific concepts of a domain (D), common concepts of multiple domains (C), and control (T). Specific concepts of a domain (D) refers to tables modeled to store information directly related to the represented domain, e.g. *medical*

procedure; common concepts of multiple domains (C) refers to tables modeled to store information where the concept is a standard to many domains, e.g. *Marital status*; and control (T), refers to tables that store information relating to several tables and store repeating facts, e.g. sales.

Table classification is performed manually in Configuration Template according to its tuples content and its classification is filled in the field *Table_type*. Tables classified as control for example, have an extensive amount of information which do not add representativeness of a domain. Meaning it is not relevant to generate instances of all database tables. Similar proposal was made by [46], which proposes a classification of database data in two parts: data of core objects in business behaviors and transactional data flow tuples in business behaviors. However, regardless of classification, instances from tuples are generated. In our approach, based on table classification we propose to map database tuples to instances or classes.

Rule 5: mapping tuples to instances

In Configuration Template the field *Related_Table* has the table name of tuples. Every tuple from tables classified as specific concepts of a domain (D) are mapped to instances. None of the analyzed solutions describe how to name ontological instances, an exception is [46] which translates primary key values to URIs postfix of instance names.

When a description attribute is not used as part of the instance name, it becomes necessary to analyze the instance property values to recognize such instance, instead of identifying it only by its name. Considering the tuple *(8310, Orthodontic Treatment)*, it is desirable that the correspondent instance in ontology carries the description *Orthodontic Treatment* in its name. Thus, a rule must be defined to set ontological names of instances to avoid conflicts when instances of different classes have the same description.

For this, the instance name is composed of: a) tuple values, whereby we recommend the value corresponding to the PK column and the column content (or columns) - with a capital letter. In case of instance name, as we can have many values, we separate the

values with underscore. The definition above was partially proposed by Tissot ² and partly to Rule S of Table 2.1. The sequence of steps of this mapping rule is displayed in Algorithm 3.7.

Algorithm 3.7: Rule 5 - mapping tuples

```

1 read configurationTemplate;
2 tuples  $\leftarrow$  configurationTemplate.type(R);
3 insert into MappingModel.T004_RECORD(tuples);
4 for ( $i = 0$ ; tuples is not null;  $i++$ ) do
5     tuple  $\leftarrow$  tuples[i];
6     for ( $i = 0$ ; tuple is not null;  $i++$ ) do
7         columnRecord  $\leftarrow$  formattingFunction(tuple[i]);
8         insert columnRecord into MappingModel.T023_COLUMN_RECORD_VALUE;
9     end
10    if tuple.RelatedTable.tableType is D then
11        insert into MappingModel.T015_INSTANCE(formattingFunction(tuple));
12    else
13        go to Algorithm 3.8;
14    end
15 end
16 Result: Mapping tuples to instances

```

Considering the database tuple (*8310, Orthodontic Treatment*) in *T008_Procedure* table, its mapping to an instance is named as *8310_Orthodontic_Treatment*, where: a) *8310* correspond to primary key column; and b) *Orthodontic Treatment* correspond to the column content. After the mapping, the elements are represented by the following OWL tags in the generated ontology:

Example 15:

```

<Declaration>
    <NamedIndividual IRI="#8310_Orthodontic_Treatment"/>
</Declaration>

```

²Tissot, Hegler Correa; Del Fabro, Marcos Didonet. Mapping relational databases to ontologies. Informatics Department, Federal University of Paraná, Brazil, 2014. Not published.

```

<ClassAssertion>
  <Class IRI="#Procedure"/>
  <NamedIndividual IRI="#8310_Orthodontic_Treatment"/>
</ClassAssertion>

```

Rule 6: mapping tuples to classes

A proportion of tables in the database have a small number of tuples which represent specific definitions of a particular domain. Considering classes are concrete representations of concepts, we classified tables with specific definitions of a domain as common concepts of multiple domains (C) and all tuples from these tables are mapped to classes, wherein the class name is the tuple content - starts with a capital letter. The definition above was partially proposed by Tissot ³. Algorithm 3.8 contains the pseudocode of this mapping rule.

Algorithm 3.8: Rule 6 - mapping tuples to classes

```

1 continue from line 14 of Algorithm 3.7;
2 if tuple.RelatedTable.tableType is C then
3   | insert into MappingModel.T011_CLASS (formattingFunction(tuple));
4 end
5 Result: Mapping tuples to classes

```

Considering the *T017_Marital_Status* table is defined as a conceptual table, the tuple $(1, Single)$ is mapped to a class named *1_Single*, where: a) *1* correspond to primary key column; and b) *Single* correspond to *Description* column. The *1_Single* class is defined as subclass of *MaritalStatus* class. After the mapping, the elements are represented by the following OWL tags in the generated ontology:

Example 16:

```

<Declaration>
  <Class IRI="#1_Single"/>
</Declaration>

```

³Tissot, Hegler Correa; Del Fabro, Marcos Didonet. Mapping relational databases to ontologies. Informatics Department, Federal University of Paraná, Brazil, 2014. Not published.

```

<SubClassOf>
  <Class IRI="#1_Single"/>
  <Class IRI="#MaritalStatus"/>
</SubClassOf>

```

Mapping check constraints

Database check constraints are a type of integrity constraint which can ensure that only specific values are allowed in the certain column. Although [9] can convert check constraints with specifically enumerated values to data range definition, it does not create a mapping which can be reused by other entities. We propose to map the check constraint restricted by a list of possible values, defining in the ontology a specification of a domain from the check constraint (term used to represent the set of values), the possible values of the check constraint and the check constraint column identification.

Rule 7: mapping check constraint concept

The *check constraint concept* represents a term which we use to represent a set of values defined in a check constraint. We propose to map the database check constraints and create a class to represent the set of values defined in the check constraint. For this, in the Configuration Template, the field *Concept_Check_Value* is filled the check constraint concept and the field *Check_Value* is filled the set of values defined for this check constraint.

For the check constraint concept we map this information to one ontological class, wherein the class name is composed of the value corresponding to the check constraint concept - with a capital letter. Every possible value of this check constraint is mapped to an instance, wherein the instance name is composed of: a) the value corresponding to the class constraint name - with lowercase letter; b) we separate the values with underscore; and c) the value corresponding to the constraint name - with a capital letter. This rule was proposed by Tissot ⁴. Following, we present Algorithm 3.9 with the pseudocode of

⁴Tissot, Hegler Correa; Del Fabro, Marcos Didonet. Mapping relational databases to ontologies. Informatics Department, Federal University of Paraná, Brazil, 2014. Not published.

this mapping rule.

Algorithm 3.9: Rule 7 - mapping check constraint concept

```

1 continue from line 16 of Algorithm 3.4;
2 class ←formattingFunction(column.SubjectCheckValue), formattingFunction(table);
3 insert table, class into MappingModel.T007_CHECK_SUBJECT,
   MappingModel.T011_CLASS;
4 values ←formattingFunction(column.CheckAbreviation, column.CheckValue);
5 for ( $i = 0$ ; values is not null;  $i++$ ) do
6   value ←values[i];
7   instance ←formattingFunction(value);
8   insert instance, class into MappingModel.T006_CHECK_VALUE,
   MappingModel.T015_INSTANCE;
9 end
10 go to Algorithm 3.10;
11 Result: Mapping check constraint concept

```

Every generated instance are associated to the class which represents the check constraint. For example, a check constraint defined to the *inactive* column is mapped to the class named *Boolean* and two instances: *boolean_True* and *boolean_False*, defined as possible constraint values.

After the mapping, as we can be seen in Example 17, the elements are represented by the following OWL tags in the generated ontology:

Example 17:

```

<Declaration>
  <Class IRI="#Boolean"/>
</Declaration>

<Declaration>
  <NamedIndividual IRI="#boolean_True"/>
</Declaration>

<ClassAssertion>
  <Class IRI="#Boolean"/>
  <NamedIndividual IRI="#boolean_True"/>
</ClassAssertion>

```

```

<Declaration>
  <NamedIndividual IRI="#boolean_False"/>
</Declaration>

<ClassAssertion>
  <Class IRI="#Boolean"/>
  <NamedIndividual IRI="#boolean_False"/>
</ClassAssertion>

```

Rule 8: mapping check constraint column

In the Configuration Template, the field checked as check constraint is mapped to one object property, wherein the object property name is composed of: a) the value corresponding to the column name - starts with lowercase letter. For this object property the *domain* and *range* are assigned, where the *domain* is specified using the class created from the table column; range is specified using the class created with the Rule 7, which represents the global concept of this set of check constraints values.

Considering the database column *A008_inactive* in *T008_Procedure*, this column is mapped to the object property named *inactive*, wherein *domain* is defined to the *Procedure* class *range* is defined to the *Boolean* class. This rule was partially proposed by Tissot ⁵.

Algorithm 3.10 shows the pseudocode of this mapping rule.

Algorithm 3.10: Rule 8 - mapping check constraint column

- 1 **continue** from line 11 of Algorithm 3.9;
 - 2 objectProp \leftarrow formattingFunction(column), column.domain, column.range;
 - 3 **insert** column, objectProp **into** MappingModel.T009_COLUMN_CHECK_VALUE,
MappingModel.T021_COLUMN_TO_OBJECT_PROPERTY,
MappingModel.T019_OBJECT_PROPERTY and
MappingModel.T022_OBJECT_PROPERTY_DOMAIN_RANGE;
 - 4 **Result:** Mapping check constraint column.
-

After the mapping, the elements are represented by the following OWL tags in the generated ontology:

⁵Tissot, Hegler Correa; Del Fabro, Marcos Didonet. Mapping relational databases to ontologies. Informatics Department, Federal University of Paraná, Brazil, 2014. Not published.

Example 18:

```

<Declaration>
  <ObjectProperty IRI="#inactive"/>
</Declaration>

<ObjectPropertyDomain>
  <ObjectProperty IRI="#inactive"/>
  <Class IRI="#Procedure"/>
</ObjectPropertyDomain>

<ObjectPropertyRange>
  <ObjectProperty IRI="#inactive"/>
  <Class IRI="#Boolean"/>
</ObjectPropertyRange>

```

Mapping hierarchies

For each related table a subclass hierarchy based on foreign keys definition is created. We also propose a hierarchy of classes based on extracted data, used to represent information of specific concept.

Rule 9: mapping inheritance relationships from tables

For each foreign key which is equivalent to the primary key in non-associative tables, the class representing the referenced table is defined as a superclass of the class representing the non-associative table. For example, *A029_Arch_Id* is the primary key in *T029_Dental_Arch* table and it is a foreign key in *T030_Dental_Segment* table. Thus, the *DentalArch* class (created from *T029_Dental_Arch* table) is a subclass of *DentalSegment* class, created from *T030_Dental_Segment* table. This rule is equivalent to Rules K and L of Table 2.1. Algorithm 3.11 contains the pseudocode of this mapping rule.

Algorithm 3.11: Rule 9 - mapping inheritance relationships from tables

```

1 continue from line 11 of Algorithm 3.6;
2 relatedTable ← column.relatedTable;
3 if relatedTable is non-associative then
4   sourceTable ← column.tableForeignKey;
5   superclass ← getClass(sourceTable);
6   class ← getClass(relatedTable);
7   insert superclass, class into MappingModel.T012_HIERARCHY;
8 end
9 Result: Mapping inheritance relationships from tables

```

After the mapping, the elements are represented by the following OWL tags in the generated ontology:

Example 19:

```

<Declaration>
  <Class IRI="#DentalArch"/>
</Declaration>

<Declaration>
  <Class IRI="#DentalSegment"/>
</Declaration>

<SubClassOf>
  <Class IRI="#DentalArch"/>
  <Class IRI="#DentalSegment"/>
</SubClassOf>

```

Rule 10: mapping table record hierarchy

In Rule 6 the mapping of all tuples to classes, from tables of common concepts of multiple domains (C), was proposed. These tables have relationships with another tables and to their relation, as we describe in Rule 9, for each related table we create a subclass hierarchy based on foreign keys definition. Creation of class hierarchy based on classes mapped from tuples of these tables is proposed in this rule.

The table record hierarchy can be mapped when both related tables belong to tables classified as (C). We performed this mapping for non-associative tables and for tables in which the foreign key columns number is not greater than one in the referenced table. This rule was partially proposed by Tissot ⁶.

To clarify this mapping, according to Rule 6, all tuples from *T029_Dental_Arch* table and *T030_Dental_Segment* table are mapped to classes. Thus, the generated classes from these table tuples are defined as a subclass of the generated class to these table. Considering the *T029_Dental_Arch* table has a relationship with *T030_Dental_Segment* table, whereby one *Dental Segment* has one *Dental Arch*, the *Dental Segment* record (*1, 1, Upper Right Posterior Segment Deciduos*) has a relation with *Arch* record, specific with (*1, Upper Dental Arch Deciduos*). Algorithm 3.12 shows the pseudocode of this mapping rule.

Algorithm 3.12: Rule 10 - mapping table record hierarchy

```

1 continue from line 8 of Algorithm 3.7;
2 if columnRecord has foreignKey and tuple.RelatedTable.tableType is C then
3   go to Algorithm 3.8 and continue from line 4;
4   class ← getClass(tuple);
5   superclass ← getClass(columnRecord.foreignKey.tuple);
6   insert superclass, class into MappingModel.T012_HIERARCHY;
7 end
8 Result: Mapping table record hierarchy

```

Faced with this, we define a record hierarchy between *1_Upper_Dental_Arch_Deciduos* and *1.1_Upper_Right_Posterior_Segment_Deciduos* record classes. After the mapping, the elements are represented by the following OWL tags in the generated ontology:

Example 20:

```

<Declaration>
  <Class IRI="#1_Upper_Dental_Arch_Deciduos"/>
</Declaration>

<Declaration>

```

⁶Tissot, Hegler Correa; Del Fabro, Marcos Didonet. Mapping relational databases to ontologies. Informatics Department, Federal University of Paraná, Brazil, 2014. Not published.

```

    <Class IRI="#1_1_Upper_Right_Posterior_Segment_Deciduos"/>
</Declaration>

<SubClassOf>
    <Class IRI="##1_Upper_Dental_Arch_Deciduos"/>
    <Class IRI="#DentalArch"/>
</SubClassOf>

<SubClassOf>
    <Class IRI="#1_1_Upper_Right_Posterior_Segment_Deciduos"/>
    <Class IRI="#DentalSegment"/>
</SubClassOf>

<SubClassOf>
    <Class IRI="#1_1_Upper_Right_Posterior_Segment_Deciduos"/>
    <Class IRI="##1_Upper_Dental_Arch_Deciduos"/>
</SubClassOf>

```

Ontology axioms

None of the analyzed solutions of Section 2.4 presented definitions about ontological aspects which are not directly related to the RDB mapping. In our work, we propose a rule definition not directly related to the database but generated from the mapping of the elements. This because in ontologies we have situations where classes should not overlap, for this, they must be explicitly defined as disjoint [4].

Rule 11: disjointness axiom

The definition of disjointness axiom about two classes states that an element cannot be an instance of both classes and the disjointness axiom is relevant to ontology validation enabling a lot of inference on an ontology. We propose in the ontology generation the definition of disjointness axiom. From the data mapped and stored in a schema generated from the Mapping Model, a class can be defined as disjoint of other class that has not been mapped as its subclass. The elements are represented by the following OWL tags:

Example 21:

```
<DisjointClasses>
  <Class IRI="#Gender"/>
  <Class IRI="#MaritalStatus"/>
</DisjointClasses>
```

As we present in Example 21, the definition of disjointness axiom says directly that nothing can be both a type of *Gender* and a type of *Marital status*.

3.1.4 Generation of ontology

Upon completion of the mapping rules, the information of database elements is mapped to ontology elements and stored an instance of the schema generated from the Mapping Model. We created a functionality to generate a file containing the information of ontological elements in the OWL language format called Generation of ontology.

This functionality generates the structure of the OWL file by generating the syntaxes from the results of the Mapping Process. The structuring of the ontology elements in the file is performed in the following order: a) classes; b) subclasses; c) datatype property, datatype property domain, datatype property range, datatype property UnionOf and subproperty; d) object property, object property domain, object property range, object property minCardinality and inverse object property; and e) instance and class assertion. The generation of the OWL file allows the extracted ontological structure to be viewed and validated in ontology editors, as well as it allows the ontology to be accessed automatically by other applications.

3.2 Prototype

We developed a Prototype to automatically perform the Mapping Process. The Prototype has a web interface and was implemented using Java programming language, using jdk 1.8.0 _77, Apache Tomcat 7.0.37 application server and MySQL 6.3.6 database. The Prototype is able to transform the information manually filled in the Configuration Tem-

plate file to OWL file. We perform the manual upload of Configuration Template in the Prototype, and it starts the execution, first reading the data filled and on sequence the Mapping Process automatically applying the mapping rules numbers 1 to 10, described in the 3.1.3.3 section. We created a schema generated from the Mapping Model, structured in a schema of Mysql database to store the processed information from the mapping. At the end of the Mapping Process execution, the Prototype generates the ontology and makes downloadable the OWL file.

In the Figure 3.5, the painted area corresponds to the components of our architecture which were developed in the Prototype.

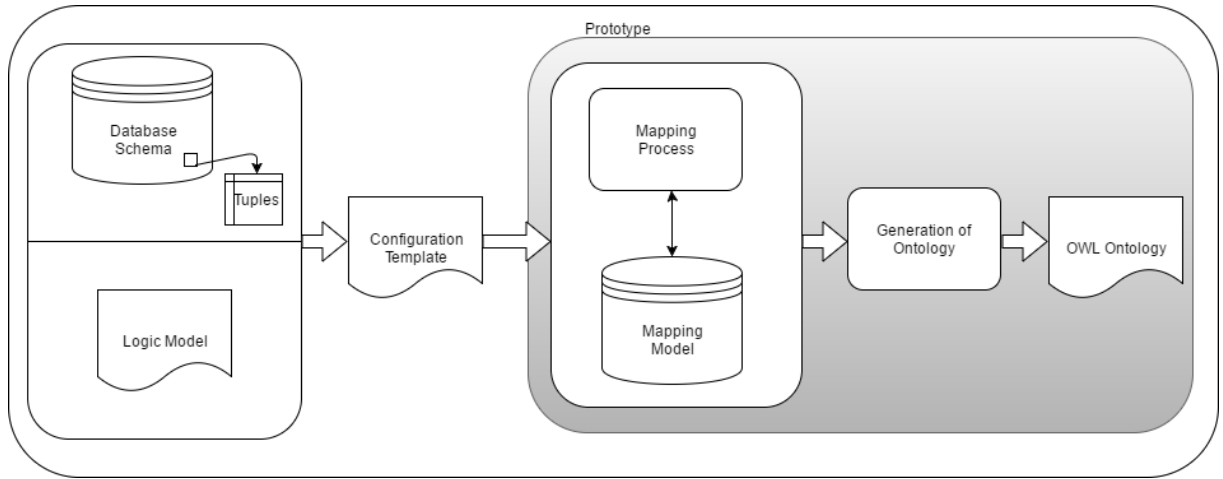


Figure 3.5: Architecture components of prototype

The Prototype is divided into structures, which are themselves subdivided into four layers: *action*, *bean*, *bo* and *dao*. In the *action* layer classes that receive the requests and guide the result of the processing are found. In the *bean* layer, the classes that define objects and their attributes, which are necessary to carry out the transformation process, are defined. In the *bo* layer the classes that execute the business rule of the Mapping Process and generation of the OWL file are found, and in the *dao* layer the classes that access objects in the instance of the schema generated from the Mapping Model are specified.

Appendix D introduces a step-by-step guide for configuration of the Prototype environment. Once the configuration of the environment is set, we need a Configuration Template file filled with the Database Schema, Tuples, and information from Logic Model,

in order to perform the Mapping Process. For the execution of the Mapping Process in the Prototype the following two pieces of information are required: the database name, information which, in addition to identifying the database that will be mapped, is used to name the ontology; and the upload of the Configuration Template file. Figure 3.6 presents the Prototype screen, which has a web interface.

Figure 3.6: Prototype

3.2.1 Development of mapping process

The Mapping Process performs different types of mappings, which occur according to the types of database elements informed in the Configuration Template file. The mapping is processed in the following sequence: database tables, database columns, database tuples, and database.

In the Prototype, the Mapping Process is the main functionality which integrating the architecture components. Through our Prototype this process comprises receiving manually the Configuration Template file; apply the first 10 mapping rules presented in subsection 3.1.3.3; and store the database and ontology elements in the instance of the schema generated from the Mapping Model. In order to do this, the Prototype reads the content of the Configuration Template and sends it to the Mapping Process which

automatically starts the mapping through the rows that have table information. After that, the lines containing the information of columns are processed and finally, the tables tuples are manually classified as specific concepts of a domain (D) and common concepts to multiple domains (C).

The functions which implement the mapping rules initiate the mapping by non-associative tables and concepts which represent the range of values of a constraint, and these elements are mapped to classes. The range of constraint values are mapped for instances, and these instances are related to the class which represents the concept of the range of constraint values. After that, five functions were defined to map the table columns for object property. The first function maps unique key columns, the second maps primary key columns of associative tables, the third maps check constraint columns, the fourth maps primary key columns of non-associative tables, and the last one maps foreign key columns. There are also two functions that were elaborated to map columns for datatype property; the first function maps columns checked in Configuration Template as description columns, and the second one maps the other columns of the database that were not classified as description; or unique, primary or foreign key. The functions that map database columns for object property or datatype property also define the domain and range of these properties, adopting the definitions established in the mapping rules. Finally, we defined two functions to map the database tuples, the first to map tuples for instances and the second to map tuples for classes.

3.2.2 Use of mapping model

The Mapping Model is a metamodel which we elaborated in order to create a schema and maintain the relations between database elements and ontology elements. During the Mapping Process, the database elements are automatically inserted into the tables defined for the database context in the instance of the schema generated from the Mapping Model. Once the database elements are stored and the mapping rules are applied, the ontology elements are generated, which are also inserted into the tables defined for the ontology context.

The relations between database elements and ontology elements is what enables source-to-target traceability. The structure which we created generates an identification of the mapping, enabling different database models to use the Prototype at the same time and the information to be maintained during the mapping can be stored each one in its respective domain. The script that enables the schema creation based on the Mapping Model is presented in Appendix E.

3.2.3 Generation of OWL ontology

After completing the Mapping Process, the Prototype enables the upload of the file containing the description of the ontological structure of the OWL language. The OWL file generator was implemented in the Prototype without the aid of frameworks and other external tools. The syntaxes were defined in the scope of this work, following the standard and the organization adopted by the Protégé [5] tool.

With the generation of the OWL file, it is possible to view the ontology manually or by using a tool that supports the edition of ontologies.

3.2.4 Deployment diagram of architecture

We elaborated a deployment diagram [22], as we can observe in Figure 3.7, to represent the physical structure of the Prototype. In the deployment diagram, the three application nodes are represented, which are: the Web Client, which consists of the environment client of the Prototype; RDB_to_onto Mapping Server, which corresponds to the application server; and the Mapping Database Server, which corresponds to the schema where the database elements and ontology elements are stored. The nodes are connected through TCP/IP communication paths. Within each node are the components which we defined to represent physical modules of code are represented.

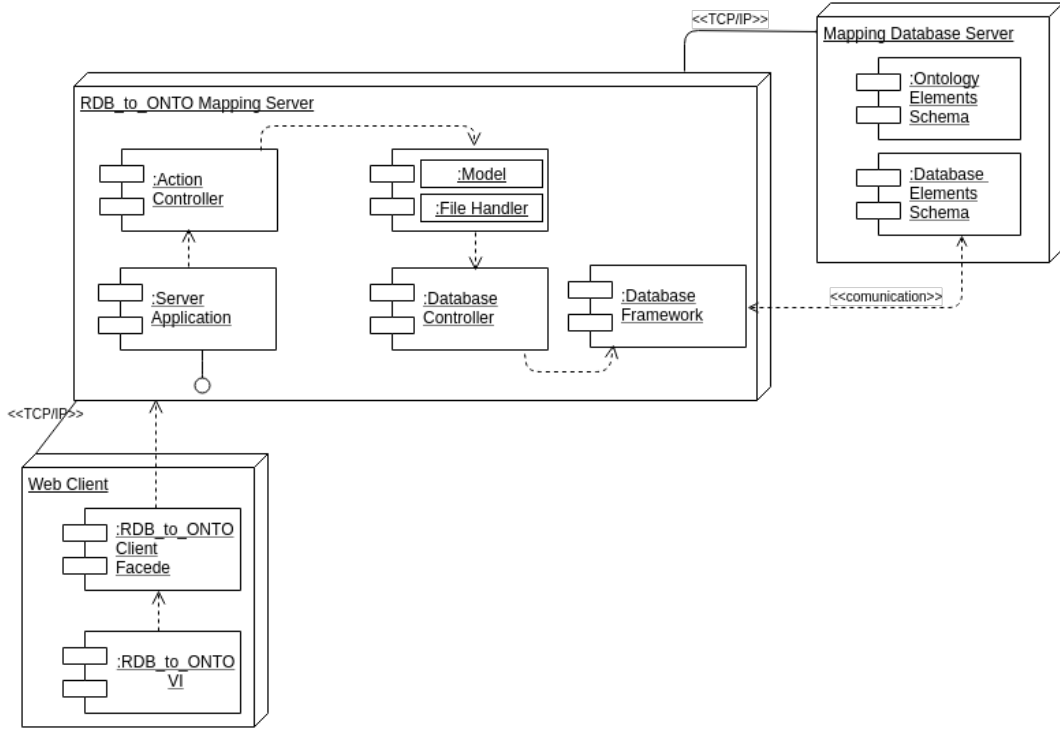


Figure 3.7: Deployment diagram of architecture

3.3 Summary

In this chapter, we present in detail each component of the architecture which we elaborated to map relational database to ontologies, but we also present in detail the Prototype developed to carry out the automatic Mapping Process.

We defined an architecture in which we can carry out the Mapping Process using from the database: a) the Database Schema, in which we can extract the database structure and the database instances; and b) the Logic Model, in which we can use in conjunction with the database structure. The database entry information definitions were defined so that the Mapping Process could be carried out with the available database resources, and it is not a deterrent to make use of the architecture in case the database has no Logic Model or, in case consideration of the database tuples is undesired. This definition makes the use of the architecture more flexible, facilitates the validation of imported data and enables the architecture to be used in different scenarios of various degrees of complexity.

Regarding the set of defined rules, we presented new ways of mapping relational database to ontologies, which count on the support of the Mapping Model so that ele-

ments representing the same concept are not duplicated and source-to-target traceability of the mapped elements takes place. We also presented criteria to name the ontology elements, which enables conventions to be established to be used by the community and which contribute to a greater legibility and understanding of the constructed ontology.

In Rule 5 we presented that our configuration template allows creating an instance with more than one database column. The composition of instance name with the PK column is an alternative to avoid conflicts when instances of different classes have the same description. However, the architecture allows informing the desired columns to be mapped, not being an obligation use the suggested standard.

We proposed Rule 11 in this dissertation and there is no definition in the literature of a disjointness axiom definition in mapping process of a database to ontology. However, Rule 11 was not implemented in the developed Prototype.

The naming specifications of the elements in the Mapping Process and the use of the Logic Model in the Configuration Template were performed with the objective of contributing to the legibility of the ontology. This is because, from the Database Schema the database elements may contain abbreviations in the composition of their name, which may cause doubts as to their definition.

CHAPTER 4

CASE STUDY

One case study with three scenarios from a dental care system was elaborated to verify the defined rules and architecture, enabling mapping relational database to ontology. Each scenario was submitted to the developed Prototype. Section 4.2 presents the details of each scenario elaborated for the case study. Section 4.3 describes the evaluation method. Section 4.4 presents rules discussions, architecture discussions, and a comparative discussion with a related work.

4.1 Objectives

The objectives of this case study are: a) to verify the rules applied in the Mapping Process in a way that all specified rules can be tested and the traceability between database elements and ontology elements was verified; b) to validate the operation of the architecture elaborated, assessing in each of the scenarios the representativeness of the modeled ontology; and c) to make a comparison between the proposed rules by one of the related works and the rules that we defined for our Mapping Process.

4.2 Scenarios

In order to carry out the case study, the scenarios were elaborated using the same database. For each scenario, we performed the verification of mapping rules numbers 1 to 10, described in the 3.1.3.3 section. We also observed the behavior of the defined architecture through the Prototype execution.

The case study was conducted with the authorization of use and data collection of a dental services provider with the prerequisite that the identity of the institution as well as the data was preserved. In this way, all the examples presented in this work were adapted

to conceal any type of identification of people and companies involved.

The difference between the three scenarios lies in the Mapping Configuration (MC), created from the filling in the Configuration Template, which was designed to receive all the information from the Database Schema and Logic Model, as detailed in the 3.1.1 section. The following differences for gathering data are considered: in the first scenario only the Database Schema; in the second scenario, the Database Schema and the Logic Model; and the third scenario comprises the Database Schema and the Logic Model, where an analysis by a specialist was performed. This specialist refers to a database administrator responsible for maintaining the database and knowledge holder of the database business model. This activity in the third scenario consisted of filtering the definitions which perform specific domain representations and of discarding the elements defined to support security, configuration, and system log matters.

For each scenario, the database structure was extracted and populated in the MC structure, considering in all scenarios 696 tuples and the same set of 25 tables presented in Appendix F. The selected tables are associative and non-associative, and contain, in its turn, columns of different data types and relationships. In the scenarios, we filled the information of tuples only from tables classified as type of specific concepts of a domain (D) and type of common concepts of multiple domains (C). The tables classified as type of control (T) was discarded of our case study.

Table 4.1 presents the number of database elements we considered in each of the scenarios. The quantities of database elements in the first and second scenario are the same, differing only in the naming of the elements, which, in the second scenario, we used the information from the Logic Model.

Table 4.1: Number of database elements

Database elements		Number of elements		
		First scenario	Second scenario	Third scenario
Tables		25	25	25
	Primary key	45	45	45
	Unique key	1	1	1
Columns	Foreign key	19	19	19
	Check constraints	28	28	28
	No constraints	225	225	74
Tuples from tables of types D and C		696	696	696
Total		1039	1039	888
Columns with logical name		0	232	140
Tables with logical name		0	25	25

First scenario

The MC file designed to conduct the first scenario verifications only considers the Database Schema (Logic Model was not considered). As we presented in Table 4.1, we used the set of 25 *Tables* which contain 318 *Columns*, consisting of: 45 *Primary keys*, 1 *Unique key*, 19 *Foreign keys*, 28 *Check constraints* and 225 *Columns with no constraints*. We considered 696 *Tuples* which are not classified as control (T), as presented in the 16 section. This is the reason that, as mapping rules 5 and 6 states, we should not import tuples of tables with this classification.

Second scenario

The MC structure is designed to separately receive the information from the Database Schema and Logic Model. Given this possibility, we elaborated different scenarios to verify the operation of the Prototype developed. The MC file for the second scenario is designed from the items put in in the first scenario, and their inclusion of the logical name of the elements, which are found in the Logical Model. In Table 4.1 we showed that, of the 1039 database elements, 318 are columns and only 232 *Contain logical name*, adding up to 257 logical names, that is, considering the logical name of the tables.

Third scenario

There are columns created in the database to support configuration issues, data security, or logs, which, in short, do not contribute to the domain representation. Given this, we defined the third scenario in which the database schema columns defined for these types of systemic controls are disregarded from the MC filling. From the MC file filled in the second scenario, we performed an analysis by an expert in the table columns. The set of tables and information related to the primary key, unique key, foreign key and check constraints were kept. The columns which support security, configuration, and system log matters are removed from this file. In Table 4.1 we demonstrate that, from items of the second scenario, 151 *Columns without constraints* which the expert considered to be unrelated to the domain representation concepts were removed. Thus, one of the differences between the third scenario and the second scenario is the number of columns, adding up to 167 columns, of which 140 *Contain logical name*.

4.3 Method

We developed an evaluation method based on four activities: 1) extraction of the database information to fill the MC; 2) prototype execution; 3) ontology verification; and 4) tabulation of results. The sequence of the activities application is presented in Figure 4.1. Each activity represented by a rectangle generates an input artifact for the next activity.

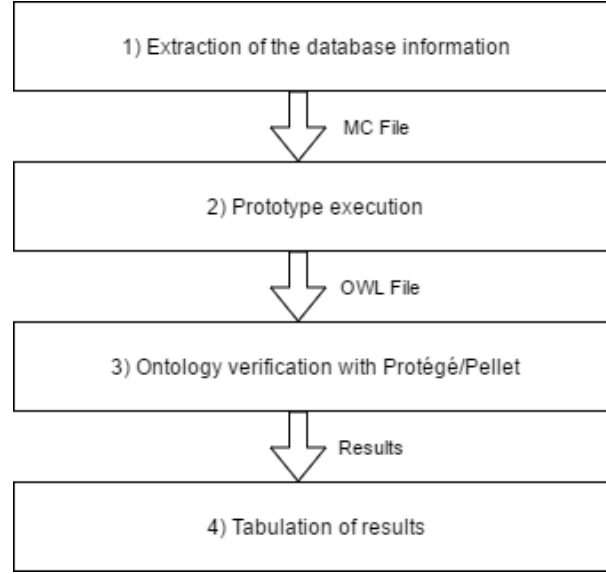


Figure 4.1: Activities of method

For activity 1) extraction of the database information (Database Schema and Logic Model) we performed a manual activity to extract the information from the Database Schema, Tuples, and Logic Model for after fill the MC file. In order to extract the information, we considered the set of selected tables in the scenarios, as described in the 4.2 section. The process of extracting the information from database tables occurs in conjunction with the filling of MC file.

We first fill in the table information, according to Appendix A, supplying in addition to the information of table structures the classification of the table representation, with the possibilities: specific concepts of a domain (D), common concepts of multiple domains (C), and control (T).

As a next step, the information regarding the columns of these tables is inserted. The source table, the data type, and the constraints are put in in each column, and for foreign key columns, the related table is also filled in. The columns which contain constraints we also supply the range of defined values, e.g. *male/female*, and a domain definition which represents them, e.g. *gender*.

According to the type of representation from the table data, we performed the extraction of table tuples, which are of specific concepts of a domain (D) and common concepts of multiple domains (C) types. The extraction of these tuples is performed through queries

carried out in the database, elaborated to individually extract the tuples of each table. The tuples were formatted and inserted into the MC. It was through this that the filling of the MC file used in the first scenario of this case study was concluded.

In the second scenario, in which we performed the filling of the second MC file, we followed the same steps as in the first scenario to extract the information from the database schema. Table and column information from Logic Model is included. While elaborating the third scenario, we initially filled the third MC file, following the same steps of the second scenario to extract the information from the Database Schema and Logic Model. The expert then performs a complete analysis of all table columns in the database schema, in order to identify and remove from the MC file the columns that were modeled to support security, configuration, and system log matters and which do not significantly contribute to the representation of the domain. In this way, the MC filling is concluded. After that, we performed activity 2) prototype execution for each of the scenarios, initiating the process by uploading the MC file. This activity was individually performed for each scenarios. When the prototype execution was completed, for each scenario an OWL file was generated.

We then performed activity 3) ontology verification in two steps: first with an analysis of mapped elements and second a verification applying inferences. Both activities were performed using the Protégé [5] tool, which is an OWL ontology creation and edition tool, developed by the Stanford University Center for Biomedical Informatics Research. Protégé allows the application of inference mechanisms in so as to verify the consistency of the definitions performed.

First, we used the Protégé to support a direct analysis of the database elements with the ontology elements generated in each of the scenarios. This analysis enabled verifying whether the defined mapping rules were applied correctly by the Mapping Process. For this reason, we analyzed all the generated elements in all the scenarios. We observed matters regarding the elements organization, the naming and specific characteristics of each one. Second, we used the Protégé to apply inferences in each OWL file using the reasoner Pellet [6]. The inferences carry out the ontology classification, the computation

of inferred instances and the validation of ontology consistency.

After that, we performed activity 4) tabulation of results in which we presented a number of ontology elements generated from the database elements considered in the three scenarios, as presented in the Section 4.2.

4.4 Results and discussions

In order to validate the operation of the architecture and the application of the rules, we submitted the three scenarios to the Prototype. Table 4.2 presents the number of ontology elements generated for each scenario.

Table 4.2: Number of ontology elements				
Ontology elements		Number of elements		
		First scenario	Second scenario	Third scenario
Class		109	109	108
	SubClass	121	121	121
Object property		68	74	73
	Functional	50	54	54
	Inverse	30	32	32
	Domain	105	107	107
	Range	84	86	85
	Min cardinality	76	76	76
Datatype property		142	122	64
	SubProperty	5	5	5
	Domain	141	128	67
	Range	142	128	66
Instances		668	668	668
	Class assertion	668	668	668

We classified the results of Table 4.2 by scenario (columns) and in four categories of ontology elements (lines), which are: *Class*, *Object property*, *Datatype property*, and *Instances*. In the *Class* category we presented the total of *Class* and *Subclass* that were generated. The elements of the *Class* category were generated from Rules 1, 2, 6, 7, 9, and 10, as covered in the 3.1.3.3 section. We also presented the total number of *Object property* elements and their definitions of *Functional*, *Inverse*, *Domain*, *Range*, and *Min cardinality* for Object property. The elements of the *Object property* category were generated from

Rules 2, 4, and 8. In the *Datatype property* category we presented the total number of Datatype property elements, and their *SubProperty*, *Domain*, and *Range* definitions. The elements of the *Datatype property* category were generated from Rules 3. Finally, we presented the total number of *Instances* and *Class assertion*, generated from Rules 5 and 7.

Based on the quantities previously presented in Table 4.1, we can observe that the number of database elements for the first and second scenarios is the same. However, due to the use of the Logic Model in the second scenario, we observed in Table 4.2 a difference in the number of ontology elements that were generated. On the one hand, the logical name can present specification which differentiates the elements, generating a greater amount, such as the elements of *Object property*, which go from 68 in the first scenario to 74 in the second scenario. On the other hand, the logical name of few elements can be the same, generating a smaller number of elements, such as the *Datatype property* elements, which go from 142 in the first scenario to 127 in the second scenario.

To provide a better understanding of the results presented in Table 4.2, we present in the Subsection 4.4.1 observations regarding the mapping rules. In Subsection 4.4.2 we cover the operation of the architecture considering the three scenarios. In Subsection 4.4.3, we present the obtained results by applying rules from another work [9], and a comparative discussion of the obtained results.

4.4.1 Rule discussions

From the set of rules we presented in 3.1.3.3 and so as in to verify the rules applied in the Mapping Process, we performed the following activities: 1) verification in parts of a set of rules that sensitize a given set of elements; and 2) verification of each database element and its respective generated element in the ontology.

Activity 1) verification in parts of a set of rules, was established so that we could test isolated mappings for the generation of each specific set of ontology elements. This is because ontology verifications in parts enable controlled observation of the generated elements, facilitating the identification of defects.

For this, we divided the Generation of Ontology into four parts. For each ontology structure verified, new ontology elements are incremented in the following order: 1) *Class* and *Subclass*, 2) *Datatype properties*, 3) *Object properties*, and 4) *Instances*. Each generated ontology, in a total of four, was individually checked with the Protégé tool [5]. This tool supports check generated elements. The reasoner Pellet [6] was used to perform classifications, compute instances, and validate the ontology consistency.

The first generated ontology was composed of *Class* and *Subclass* which derived from Rules 1, 2, 6, 7, 9, and 10, as presented in the 3.1.3.3 section. By using Protégé we were able to view the ontology and we identified that the *Class* and *Subclass* were correctly declared. We then executed the Pellet reasoner and there was no identification of inference errors in the ontology.

The second ontology was composed of *Class*, *Subclass* and we added *Datatype properties*, which derived from Rule 3. In the ontology, we observed that the elements of *Datatype properties* and the definitions of *SubProperty*, *Domain* and *Range*, were represented correctly. We executed the Pellet reasoner, and the inferences in the ontology were carried out successfully.

The third ontology was structured with *Class*, *Subclass*, *Datatype properties* and we added *Object properties*, generated from Rules 2, 4, and 8. The elements of *Object properties* and the definitions of *Functional*, *Inverse*, *Domain*, *Range* and *Min cardinality* were represented correctly, and the inferences were successfully applied.

The fourth and last ontology was composed of *Class*, *Subclass*, *Datatype properties*, *Object properties* and we added *Instances* and *Class assertion*, generated from Rules 5 and 7. The generated ontology represented correctly all the elements, and the inferences were applied successfully.

After the ontology verification in parts, we generated an ontology for each single one of the scenarios. We identified the representation of all mapped elements in each scenario and the inferences were successfully applied. Figure 4.2 presents an example of class hierarchy classification inferred by the reasoner Pellet in the third scenario. In the left part (a) of the Figure, we present the asserted hierarchy of the Classes, which corresponds to the

hierarchical representation in which the Classes were initially constructed. In the right part (b) of the Figure, we present the hierarchy of Classes inferred and automatically calculated by the reasoner Pellet. Regarding the class equivalence classifications, no inferences were made because mapping rules do not include definitions of equivalence.



Figure 4.2: Class hierarchy classification: a) asserted and b) inferred

In Figure 4.3 we present an illustrated example of an analysis performed. Give the following database elements: *T002_Dentist* (*a002_dentist_id*: int, *a002_cro*: varchar, *a002_dentist_name*: varchar, *a017_marital_status_id*: int) with the Mapping Process we obtained the following elements of ontology: *Dentist* (*cro*, *dentistName*, *hasDentistId*, *isDentistIdOf*, *hasMaritalStatusId*, *isMaritalStatusIsOf*).

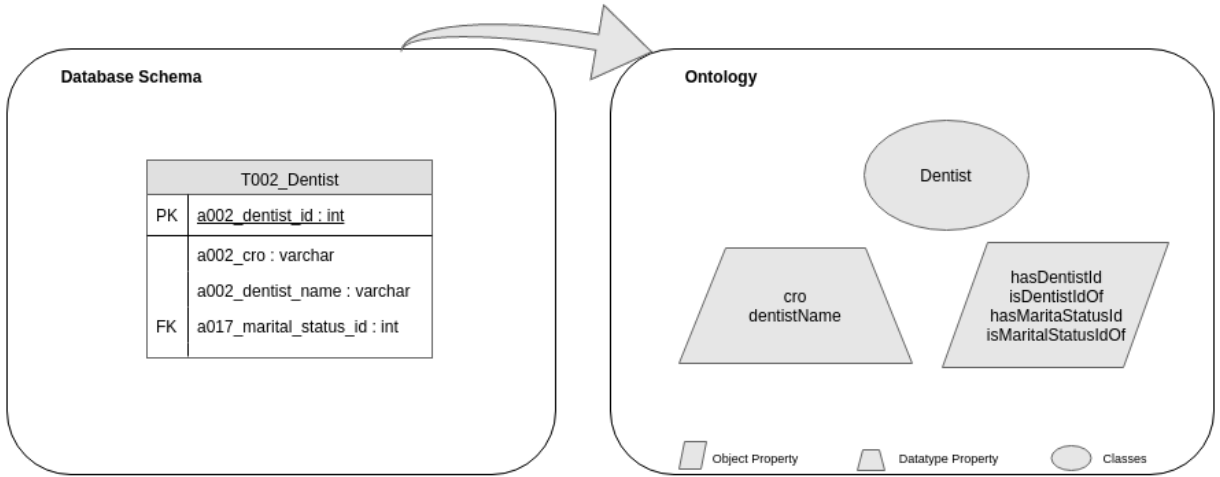


Figure 4.3: Example of mapping rules

We can observe an example in Figure 4.3 in which Rules 1, 3 and 4 were applied. Rule 1 maps non-associative table to class, where table *T002_Dentist* is mapped for a class *Dentist*. Rule 3 maps database columns to Datatype properties, where columns *a002_cro* and *a002_dentist_name* are mapped to the Datatype properties *cro* and *dentistName*. Rule 4 maps database columns (PK, FK e UK) to Object properties, where

columns *a002_dentist_id* and *a017_marital_status_id* are mapped to the Object properties *hasDentistId*, *isDentistIdOf*, *hasMaritalStatusId*, and *isMaritalStatusIsOf*.

When we individually check each of the rules and each generated element in the ontologies of the three scenarios, we verify: the fulfillment of the definitions established in each one of the Mapping rules; the naming and correct application of specific characteristics of each element; the correct grouping of elements which could generate duplicity; and the establishment of relations among the database elements in the instance of the schema generated from the Mapping Model.

4.4.2 Architecture discussions

The architecture validation was performed in each of the architecture components, as presented in the 3.1 section. We observed the behavior of the architecture and the ontology representativeness in each of the scenarios.

The Configuration Template structure fulfilled satisfactorily the filling of the database elements consumed by the Mapping Process. The developed Prototype processed the Mapping Configuration (MC) and all elements declared in the file were mapped. We had few problems with tuples when part of their content contained quotation marks (“”), requiring the elimination of those characters from the file. We observed in few tuples the existence of other characters such as: (? , Ç, -, *, *); which, however, did not cause errors in the execution of the Prototype.

We verified the implemented Rules in the Prototype, both in respect to the coherence of the rule definitions, as we covered in the Subsection 4.4.1, and regarding the mapping of the elements in the instance of the schema generated from the Mapping Model. In order to verify the mapping of elements, we query the tables and we verified whether all the database elements reported in the MC were stored in the instance and related to the ontology elements. As for the Generation of Ontology, the OWL file was constructed correctly with all mapped ontology elements.

We noticed that the naming of the elements in the Mapping Process and the use of the Logic Model in the Configuration Template increase the legibility of the ontology.

For instance, in the three scenarios, there is the *Column a002_ci_Identi*, considering the application of the naming specifications in this element, the numbers, and special characters are removed, mapping according to Rule 3, *Datatype property CiIdenti*. However, the name of the element causes a fuzzy understanding of its correct representation, being either *Civil identity* or *Citizen identification*. With this, we confirmed that legibility for humans is impaired when we perform Mapping Process using only database schema. With the Logic Model, we identified that the *Column a002_ci_Identi* has the logical name *Number of civil identity*. When we considered this logical name in the Mapping Process, we map this *Column* to the *Datatype property NumberCivilIdentity*, which is more descriptive than *CiIdenti*, generated from the Database Schema.

Therefore, we observed a greater clarity in the element names of the second scenario, that is, compared to the first scenario, highlighting one of the benefits gained through the use of the Logic Model in the Mapping Process. With the elaboration of the third scenario using the Database Schema, Logic Model, and the analysis of an expert in the database elements to be mapped, we obtained, besides a representation of the domain with greater legibility, definitions which are more coherent to the domain. In the first and second scenarios, several mapped elements, such as columns defined to ensure data security, generated a large amount of information which was not related to the domain representation. During the analysis of the expert in the third scenario, out of the 151 *Columns* that were excluded, 75 controlled the security of database tuples and 76 controlled operations of the system which populates data in the database.

The Mapping Model enabled us to map the relation between the database elements and the ontology elements. Unlike other approaches which performed only 1:1 mapping of the elements, we modeled a database schema from the Mapping Model so that 1:N relationships could be represented. Due to this structure, n database elements can be mapped to 1 element in the ontology, and vice versa. A practical example is the representation of a property generated from a primary key column. This column may exist in different tables, being represented by a foreign key column. We understand that it does not make sense to represent n times the same element that represents the same concept

but inserted in different domains.

Table 4.1 shows that in the three scenarios there are 93 database elements classified as *Primary key (PK)*, *Unique key (UK)*, *Foreign Key (FK)* and *Check constraints*. These elements, according to Rule 4, are mapped to the *Object properties*. When looking at the results presented in Table 4.2, for the first scenario 68 *Object properties* were generated, while in the second scenario 74 and in the third scenario 73. In these three scenarios, the amount of *Object properties* is smaller than the 93 database elements, showing the elimination of elements that would be duplicated in the ontology. This matter is made clear when we identify in the *Object properties* several *Domain* definitions, each being specified to represent the different domains in which the *Object property* is inserted, e.g. the *Object property hasCityId*, where the source of the element is a *PK*, containing the *Range City* and the *Domains Dentist, Patient, and Neighborhood, Classes* which represent tables where this *PK* is an *FK*.

Unlike other proposals which map classes from one table, Rules 1, 2, 6, and 7 set criteria for mapping different types of database elements to classes. With this, as seen in Table 4.2, we could represent a greater number of *Classes* in the ontology, that is, compared to the number of *Tables*, as can be seen in Table 4.1. Although the number of database elements which are mapped to *Classes* is the same in all three scenarios, the third scenario presents one *Class* less because, during the expert analysis, a column was removed from an associative table. With this, a Rule 2 criterion was not applied, which generates a *Class* for non-associative tables which contain columns other than primary keys, thus generating one *Class* less than in the first and second scenarios.

Although we considered the third scenario to be the most interesting for the construction of domain ontology and, in this sense, the expert analysis is indispensable and consists of a manual activity, the automatic extraction of the database schema is an issue that would quicken the process of the MC filling, even whether adjustments were made into the file structure.

4.4.3 Comparative discussion

The mapping process defined by Irina [9] is one of the proposals which clearly presents the definition and application of Mapping rules, proposing rules for mapping tables, columns, constraints, and tuples. Although [9] mentions an electronic address to access the Prototype developed for the QUALEG DB tool, such address is no longer available for online access.

The database elements used in the three scenarios and totaled in Table 4.1 were used to simulate the application of the mapping rules proposed by [9]. For this, we performed the following activities: 1) gathering of the proposed rules in [9]; 2) simulating the mapping process of Astrova [9], which we refer to as process B; and 3) tabulating the obtained results.

For activity 1) gathering of the proposed rules in [9], where 13 rules were identified, 3 of them to map *Tables*, 9 to map *Columns* and 1 to map *Tuples*. After this, activity 2) simulating the mapping process of [9] was conducted. For each rule we identified the number of database elements, we simulated the mapping process according to the rules gathered and took note of the number of ontology elements which would be generated according to process B. Finally, we performed activity 3) tabulating the obtained results, where we totaled the number of elements which would be generated according to process B. Table 4.3 presents the number of elements generated in each of the scenarios, where each scenario was related to the results of our mapping process, which we refer to as A and the simulation results of the mapping process.

Table 4.3: Comparative of generated ontology elements

Ontology elements		Number of elements					
		First scenario		Second scenario		Third scenario	
		A	B	A	B	A	B
Class		109	24	109	24	108	24
	SubClass	121	19	121	19	121	19
Object property		68	65	74	65	73	65
	Functional	50	23	54	23	54	23
	Inverse	30	27	32	27	32	27
	Domain	105	42	107	42	107	42
	Range	84	42	86	42	85	42
	Min cardinality	76	42	76	42	76	42
Datatype property		142	225	122	225	64	74
	Enumerated	0	25	0	25	0	25
	SubProperty	5	0	5	0	5	0
	Domain	141	225	128	225	67	74
	Range	142	225	128	225	66	74
	Max cardinality	0	225	0	225	0	74
Instances		668	221328	668	221328	668	221328
	Class assertion	668	221328	668	221328	668	221328

A= Results of our mapping rules

B= Results of mapping rules proposed in [9]

Based on the data presented in the simulation of the mapping process B, we performed comparative analyses. Rules of B generate the *Enumerated* and *Max cardinality* definitions for the *Datatype property* element, unlike the definitions we made. However, we defined *SubProperty* for the element *Datatype property* and we observed that due to the definitions for mapping tuples of B generate many *Instances*, where A=668 and B=221328; consequently the ontology will contain the representation of many elements. Regarding this, Rule 5, which we defined and presented in the 3.1.3.3 section for mapping instances, does not generate elements for each single piece of the information represented in a tuple. We generated in A only one instance for all tuple content, therefore, for process A of Table 4.3, the amount of *Tuples* and *Instances* is the same. Considering that a generated ontology can be used to share a common understanding of the domain represented in it, a smaller ontology is easier to be processed and to be interpreted by people.

Rules 6, 7, and 8 results in an increase in the number of generated elements. We noticed that our Mapping Process (A) generates a greater number of *Class*, *Subclass* and

Object properties. We concluded that, the greater number of *Class*, more definitions of concepts are represented in the ontology; the greater number of *Subclass*, the greater number of taxonomies which represent more sets and subsets of elements; and the greater number of *Object property*, more associations among concepts are represented.

For the effect of separating the content which represents knowledge domain from the content which represents operational knowledge, in the Rules defined in this paper and presented in the 3.1.3.3 section, we proposed methods to address this matter. The tuples mapping is not applied to all tables. The tables are classified precisely to map the information in it stored and to conduct the mapping only of data related to the represented domain.

4.5 Summary

The three scenarios enabled us to observe the behavior of our architecture in the light of a) the process of mapping performed only with the database schema; b) the Database Schema with the Logic Model, or, c) the Database Schema with the Logic Model and the analysis of an expert. We noticed that when the mapping process with the logical name of the database elements is carried out, it results in a more legible ontology, that is, compared to the Mapping Process considering only the database schema. Through the case study, it was also possible to evaluate the operation of the architecture in the Mapping Process, the traceability between the mapped elements and the defined mapping rules.

Given the results of the third scenario, compared to the second scenario, we came to understand that expert analysis in the Configuration Template contributes to the Mapping Process. This is due to the information which is modeled in the database, but which do not perform the representation of a concept of the mapped domain, being irrelevant for the mapping process.

So as in to simplify and standardize the name of the elements of the ontology, we applied a function to format the name of the elements, in order to maintain a naming pattern. In case the logical name is not entered in Mapping Configuration, we also apply

the formatting function to the physical name. Following the principle of elimination of duplicated elements, we elaborated mechanisms for the non-generation of duplicated elements and so that a hierarchy relation among these items is established, as well as a single definition of concepts.

We observed a greater number of ontology elements to define concepts, taxonomies, and associations through the comparative analysis of our Mapping Process (A), with the simulation of the mapping process B in [9].

CHAPTER 5

CONCLUSIONS

Ontologies define classes, properties, relations, restrictions and axioms on a given domain of knowledge, to represent elements of the real and conceptual world. Through the use of ontologies, a specific domain can be modeled and used in several ways. We understand that an ontology is only a start for a series of actions that follow. With it we can direct efforts for the development of semantic web applications, aiming at obtaining more accurate results in information research. We can use it in order to obtain interoperability, detect inconsistencies and define terminology standards.

Considering the large amount of database that we have been modeling and being used by systems, we have proposed in this work an architecture to construct ontologies of specific domains of knowledge from a relational database. We designed an architecture based on four main definitions: a) we designed a schema to map the relations between concepts of relational database and ontologies; b) we defined a Configuration Template which can be filled with the Database Schema, Tuples and Logic Model; c) we developed a prototype to read the Configuration Template, to perform the Mapping Process and to store the mapping results in a an instance of the schema generated from the Mapping Model; and d) Prototype reads the data stored in the instance to generate the OWL file.

By filling the Configuration Template of our architecture, we identified the importance of an expert analysis of the information which is filled in. It enables the complementation of Logic Model definitions and even enables to check whether all database elements should be considered by the Mapping Process. Examples of complementation of information are: a) inclusion of constraints which were not defined in the conception of database schema, and b) exclusion of elements which have been defined in the database schema, but they do not correspond to conceptual definitions of the domain to be represented in the ontology. In this way, we realize the filling of the Configuration Template, generating the Mapping

Configuration, an activity which, even it demands a great effort to be performed manually, when carried out in parallel with the analysis of the content by a specialist, results in an ontology with greater legibility and definitions which are more consistent with the domain represented.

We observed the need of maintaining the mapping between relational database elements and ontology elements and with this have the traceability of the elements, aiming to eliminate duplicate ontology elements which have the same meaning in the mapped domain. With a mapping structure we can expand our possibilities, e.g. a) we can track the transformation process for its verification; b) we can perform a reverse engineering from ontology to relational database; c) we can design a software for querying data from relational database through ontology; and others features which mapping structure can facilitate. Therefore, we elaborate a Mapping Model, integrated to the architecture. The Mapping Model, in addition to making possible the benefits mentioned above, it supported the validation process of the mapping rules.

When we conducted an analysis of previous works, we revised mapping rules and we do not identify a preoccupation about how ontology elements should be named. We also do not identify works which use the Logic Model as an input to the mapping process. The Logic Model contains definitions of objects or events which occurring in the real world and its definitions can enrich the name of ontology elements. In light of this, in our architecture, in addition to the database schema, we consider using the Logic Model for gathering more details of database elements. Based on the analysis of previous works in the area, we rewrite the proposed mapping rules for better naming the ontology elements and for correcting mistakes identified. We consider that the goal of designing an architecture to map the relational database to ontologies has been achieved. Through the prototype developed and the case study which we carried out, we validated the mapping rules which we proposed and the operation of the architecture. The use of the Logic Model in the Mapping Process is unprecedented and contributes to the generation of a more legible ontology. The Mapping Model contributed to the traceability of the Mapping Process and the elimination of duplicate elements. These contributions had not been considered

previously and collaborate in a significant way for an adequate modeling of the elements declared in the ontology.

5.1 Future work

- Incorporate rules into the mapping process which allow to generate an ontology with definitions which complete and associate the represented concepts. For this, we propose to implement Rule 11 to define disjointness axiom in OWL ontology.
- Define disjoint classes, equivalence classes, and new rules to improve the semantics of elements. By doing this, we will be able to perform queries in the ontology which answer questions.
- We have conducted a case study which involved a manual analysis of a database. We propose to automate the extraction of Database Schema and Logic Model to quicken the process of Configuration Template filling.
- Adapt the architecture to integrate ontologies of the same domain and to generate a single ontology, establishing interoperability among different database schemas.
- Design a structure to store specific conditions of the database mapping. When the mapping is reprocessed, the architecture will automatically be able to apply these conditions.
- We propose to integrate the Mapping Process with a thesaurus for naming the ontology elements with more appropriate terms used in the domain. Thus, we propose to perform an adequacy in names of mapped ontology elements.

REFERENCES

- [1] Owl web ontology language guide. <http://www.w3.org/TR/owl-guide/>. Accessed: Jun, 2015.
- [2] Owl web ontology language overview. <http://www.w3.org/TR/owl-features/>. Accessed: Jun, 2015.
- [3] Owl web ontology language reference. <http://www.w3.org/TR/owl-ref/>. Accessed: Jul, 2015.
- [4] Protégé ontology editor. http://protege.stanford.edu/conference/2004/slides/6.1_Horridge_CommonErrorsInOWL.pdf. Accessed: Set, 2016.
- [5] Protégé ontology editor. <http://protege.stanford.edu/products.php>. Accessed: Jul, 2015.
- [6] Protege-owl reasoning api. <http://protegewiki.stanford.edu/wiki/ProtegeReasonerAPI>. Accessed: Oct, 2016.
- [7] Serge Abiteboul, Richard Hull, e Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.
- [8] Vinícius Camargo Andrade. Transformação de modelos de diagrama de sequência uml contemplando restrições de tempo e energia para rede de petri temporal. 2013.
- [9] Irina Astrova. Rules for mapping sql relational databases to owl ontologies. *Metadata and Semantics*, pages 415–424. Springer, 2009.
- [10] Irina Astrova, Nahum Korda, e Ahto Kalja. Rule-based transformation of sql relational databases to owl ontologies. *In Proceedings of the 2nd International Conference on Metadata & Semantics Research*, 2007.

- [11] Jean Bézivin e Olivier Gerbé. Towards a precise definition of the omg/mda framework. *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 273–280. IEEE, 2001.
- [12] Francesca Bugiotti. *A model oriented approach to heterogeneity*. PhD thesis, Roma Tre University, Dept. of Informatics and Automation, Roma, 2012.
- [13] Guntars Būmans e Kārlis Čerāns. Rdb2owl: A practical approach for transforming rdb data into rdf/owl. *Proceedings of the 6th International Conference on Semantic Systems, I-SEMANTICS '10*, pages 25:1–25:3, New York, NY, USA, 2010. ACM.
- [14] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.
- [15] Isabel F Cruz e Huiyong Xiao. The role of ontologies in data integration. *Engineering intelligent systems for electrical engineering and communications*, 13(4):245, 2005.
- [16] Nadine Cullot, Raji Ghawi, e Kokou Yétongnon. Db2owl: A tool for automatic database-to-ontology mapping. *SEBD*, pages 491–494, 2007.
- [17] Mona Dadjoo e Esmaeil Kheirkhah. An approach for transforming of relational databases to owl ontology. *arXiv preprint arXiv:1502.05844*, 2015.
- [18] John Davies, Dieter Fensel, e Frank Van Harmelen. *Towards the semantic web: ontology-driven knowledge management*. John Wiley & Sons, 2003.
- [19] Marcos Didonet del Fabro. *Gestion de métadonnées utilisant tissage et transformation de modèles*. PhD thesis, Université de Nantes, 2010.
- [20] Carlos Julian Menezes Araújo e Robson do Nascimento Fidalgo. Metamodelo para banco de dados.
- [21] Ramez Elmasri e Shamkant Navathe. Fundamentals of database systems sixth edition pearson education. *Reproduced with permission of the copyright owner. Further reproduction prohibited without permission*, 2011.

- [22] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.
- [23] Frederico Freitas e Stefan Schulz. Ontologias, web semântica e saúde. *Revista Eletrônica de Comunicação, Informação & Inovação em Saúde*, 3(1), 2009.
- [24] Dragan Gašević, Dragan Djuric, e Vladan Devedžic. *Model driven engineering and ontology development*. Springer Science & Business Media, 2009.
- [25] Noredine Gherabi, Khaoula Addakiri, e Mohamed Bahaj. Mapping relational database into owl structure with data semantic preservation. *CoRR*, abs/1205.5922, 2012.
- [26] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220, junho de 1993.
- [27] Thomas R Gruber. Toward principles for the design of ontologies used for knowledge sharing? *International journal of human-computer studies*, 43(5):907–928, 1995.
- [28] Nicola Guarino. Formal ontology, conceptual analysis and knowledge representation. *International journal of human-computer studies*, 43(5):625–640, 1995.
- [29] Celio Cardoso Guimaraes. *Fundamentos de bancos de dados: modelagem, projeto de linguagem SQL*. Ed. da Unicamp, 2003.
- [30] Roberto Heinzle. *Um modelo de engenharia do conhecimento para sistemas de apoio a decisão com recursos para raciocínio abduativo*. PhD thesis, Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Engenharia e Gestão do Conhecimento, Florianópolis, 2011.
- [31] Matthew Horridge, Holger Knublauch, Alan Rector, Robert Stevens, e Chris Wroe. A practical guide to building owl ontologies using the protégé-owl plugin and co-ode tools edition 1.0. *University of Manchester*, 2004.
- [32] Michal Laclavik. Rdb2onto: Relational database data to ontology individuals mapping. 2006.

- [33] Tony Lawson. A conception of ontology. *Unpublished manuscript, University of Cambridge*, 2004.
- [34] Man Li, Xiaoyong Du, e Shan Wang. A semi-automatic ontology acquisition method for the semantic web. *Advances in Web-Age Information Management*, pages 209–220. Springer, 2005.
- [35] Haiyun Ling e Shufeng Zhou. Mapping relational databases into owl ontology. *International Journal of Engineering and Technology*, 5(6), 2013.
- [36] Mohammed Reda Chbihi Louhdi, Hicham Behja, e Said Ouatik El Alaoui. Transformation rules for building owl ontologies from relational databases. *Computer Science & Information Technology (CS & IT)*, 3:271–283, 2013.
- [37] Giansalvatore Mecca, Guillem Rull, Donatello Santoro, e Ernest Teniente. Semantic-based mappings. *International Conference on Conceptual Modeling*, pages 255–269. Springer, 2013.
- [38] Giansalvatore Mecca, Guillem Rull, Donatello Santoro, e Ernest Teniente. Ontology-based mappings. *Data & Knowledge Engineering*, 98:8–29, 2015.
- [39] Alexandra Moreira, Lídia Alvarenga, e Alcione de Paiva Oliveira. O nível do conhecimento e os instrumentos de representação: tesouros e ontologias. *DataGramaZero-Revista de Ciência da Informação*, 5(6), 2004.
- [40] Muhammad Mughees. Data migration from standard sql to nosql. 2014.
- [41] Teresa Podsiadły-Marczykowska, Tomasz Gambin, e Rafał Zawiaślak. Rule-based algorithm transforming owl ontology into relational database. *Beyond Databases, Architectures, and Structures*, pages 148–159. Springer, 2014.
- [42] Alex Mateus Porn. Teste de mutação para ontologias owl. Master’s thesis, 2014.
- [43] Erhard Rahm e Philip A Bernstein. A survey of approaches to automatic schema matching. *the VLDB Journal*, 10(4):334–350, 2001.

- [44] Raghu Ramakrishnan e Johannes Gehrke. Database management systems. 2003.
- [45] C Ramathilagam e ML Valarmathi. A framework for owl dl based ontology construction from relational database using mapping and semantic rules. *International Journal of Computer Applications*, 76(17):31–37, 2013.
- [46] Yutao Ren, Lihong Jiang, Fenglin Bu, e Hongming Cai. Rules and implementation for generating ontology from relational database. *Cloud and Green Computing (CGC), 2012 Second International Conference on*, pages 237–244. IEEE, 2012.
- [47] Andrea Rodacki. Aplicação de estratégias de integração de bancos de dados: Um estudo de caso. Master’s thesis, Universidade Federal do Paraná, 2000.
- [48] Valdemar Waingort Setzer. *Banco de dados: conceitos, modelos, gerenciadores, projeto logico, projeto fisico*. Edgard Blucher, 1989.
- [49] Abraham Silberschatz, Henry F Korth, S Sudarshan, e Daniel Vieira. *Sistema de banco de dados*. Elsevier, 2006.
- [50] John F. Sowa. Ontology. <http://www.jfsowa.com/ontology/index.htm>. Accessed: Jul, 2015.
- [51] Steffen Staab e Rudi Studer. *Handbook on ontologies*. Springer Science & Business Media, 2013.
- [52] Zdenka Telnarova. Relational database as a source of ontology creation. *IMCSIT*, pages 135–139, 2010.
- [53] Mike Uschold e Michael Gruninger. Ontologies: Principles, methods and applications. *The knowledge engineering review*, 11(02):93–136, 1996.
- [54] Konstantinos N. Vavliakis, Theofanis K. Grollios, e Pericles A. Mitkas. Rdote - transforming relational databases into semantic web data. Axel Polleres e Huajun Chen, editors, *ISWC Posters&Demos*, volume 658 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010.

- [55] Denny Vrandečić. Ontology evaluation. *Handbook on Ontologies*, pages 293–313. Springer, 2009.
- [56] Denny Vrandecic, Johanna Völker, Peter Haase, Duc Thanh Tran, e Philipp Cimiano. A metamodel for annotations of ontology elements in owl dl. *Proceedings of the 2nd Workshop on Ontologies and Meta-Modeling, WoMM 2006. LNI, 96*, 2006.
- [57] Holger Wache, Thomas Voegelé, Ubbo Visser, Heiner Stuckenschmidt, Gerhard Schuster, Holger Neumann, e Sebastian Hübner. Ontology-based integration of information-a survey of existing approaches. *IJCAI-01 workshop: ontologies and information sharing*, volume 2001, pages 108–117. Citeseer, 2001.
- [58] Dewi Wisnu Wardani e Josef Küng. Mapping rdb to rdf with higher semantic abstraction. pages 59–67, 2016.
- [59] Daya C Wimalasuriya e Dejing Dou. Using multiple ontologies in information extraction. *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 235–244. ACM, 2009.
- [60] Lei Zhang e Jing Li. Automatic generation of ontology based on database. *Journal of Computational Information Systems*, 7:4:1148–1154, 2011.

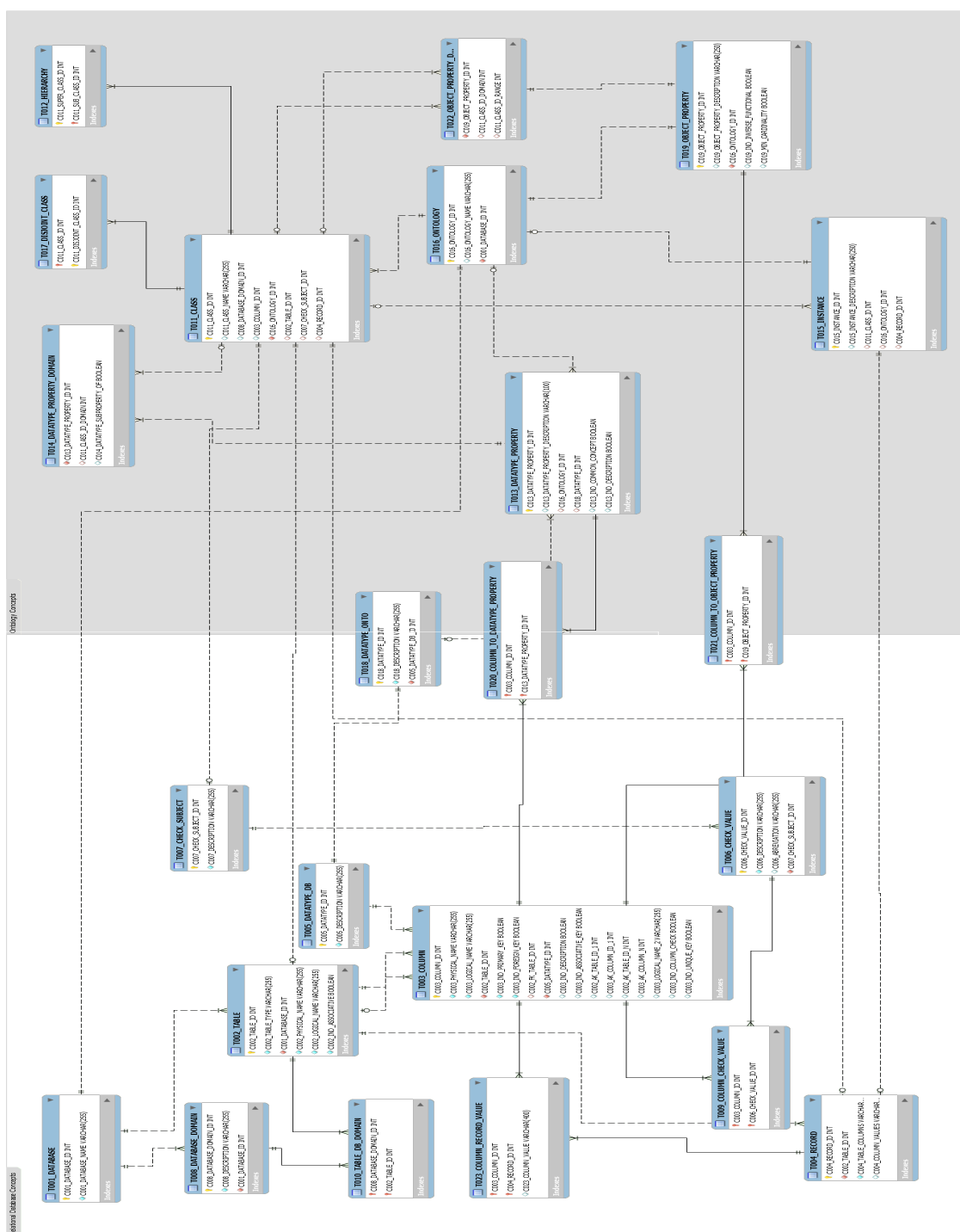
APPENDIX A

STRUCTURE OF CONFIGURATION TEMPLATE

Configuration Template Item	Used to
TYPE	In this field is filled the database element. The column receive this three value options: T(for tables); C(for columns) ; and R (for tuples). Mandatory: yes
NM.PHYSICAL.ELEMENT	In this field is filled the element physical name. Mandatory: yes
NM.LOGICAL.ELEMENT	In this field is filled the element logical name. Mandatory: no
TABLE.TYPE	In this field is filled the table type, a classification used to classify the table content. The column receive this three value options: C(as conceptual); D(as domain); and T(as transactional). Mandatory: yes - for tables
IS.ASSOCIATIVE	In this field is filled a table characteristic: 1(for associative tables) and 0(for non-associative tables). Mandatory: yes - for tables
RELATED.TABLE	In this field is filled the table which the column belongs to. Mandatory - yes for columns and tuples
COLUMN.DATATYPE	In this field is filled the column datatype. Mandatory - yes for columns
IS.PRIMARY.KEY	In this field is filled a column characteristic: 1 for primary key columns. Mandatory: no
IS.UNIQUE.KEY	In this field is filled a column characteristic: 1 for unique key columns. Mandatory: no
IS.FOREIGN.KEY	In this field is filled a column characteristic: 1 for foreign key columns. Mandatory: no
TABLE.FOREIGN.KEY	In this field is filled the foreign key source table. It is required when filled IS.FOREIGN.KEY column. Mandatory: no
IS.DESCRPTION	In this field is filled a column characteristic: 1 for description columns. Mandatory: no
IS.COLUMN.CHECK	In this field is filled a column characteristic: 1 for check constraint columns. Mandatory: no
CHECK.VALUE	In this field is filled the set of values defined for the check constraint column. It is required when filled IS.COLUMN.CHECK column. Mandatory: no
CHECK.ABREVIATION	In this field is filled the set of abbreviation values defined for the check constraint column. The values are separated by commas. It is required when filled IS.COLUMN.CHECK column. Mandatory: no
SUBJECT.CHECK.VALUE	In this field is filled the term which we use to represent a set of values defined in a check constraint. The values are separated by commas. It is required when filled IS.COLUMN.CHECK column Mandatory: no
TABLE.COLUMNS	In this field is filled the table columns inside brackets and separated by commas. Mandatory: yes for tuples
COLUMN.VALUES	In this field is filled the record values inside brackets and separated by commas. The values needs to respect the columns sequence filled in the field Table.Columns Mandatory: yes for tuples

APPENDIX B

ER MODEL OF MAPPING SCHEMA



APPENDIX C

DETAILS OF MAPPING SCHEMA TABLES

Tables of mapping schema	Used to
T001_database	stores the imported schema name.
T002_table	stores table informations.
T003_column	stores column informations.
T004_record	stores tuple informations.
T005_datatype_db	stores data types of database schema.
T006_check_value	stores values of check constraints.
T007_check_subject	stores the concept of a range of values of check constraints.
T008_database_domain	stores database definitions of domains that group tables.
T009_column_check_value	stores the relationship of columns with a set of check constraint values.
T010_table_db_domain	stores the relationship of tables with database definition of domain.
T011_class	stores the mapped classes.
T012_hierarchy	stores the mapped subclasses of a class.
T013_datatype_property	stores the mapped datatype properties.
T014_datatype_property_domain	stores the domain and range of datatype properties.
T015_instance	stores the mapped instances.
T016_ontology	stores the created ontology to the mapped database schema.
T017_disjoint_class	stores the mapped disjoint classes of a class.
T018_datatype_onto	stores data type of ontology datatype properties.
T019_object_property	stores the mapped object properties.
t020_column_to_datatype_property	stores the relationship of each column to each datatype property.
t021_column_to_object_property	stores the relationship of each column to each object property.
t022_object_property_domain_range	stores the domain and range of object properties.
t023_column_record_value	stores the relationship of each value of tuple with corresponding column.

APPENDIX D

PROTOTYPE CONFIGURATION

Technologies used:

MySQL version 5.0.8

Apache Tomcat version 7.0.37

Java version 8

Struts version 1.3.10

Hibernate version 5

Eclipse (version: Neon Milestone 1 (4.6.0M1)):

Installation:

1 - Through Git CMD, access the workspace folder that will be used by Eclipse.

2 - Execute the following command:

```
git clone https://github.com/caghuve/rdb-to-onto.git
```

3 - Open Eclipse and create a new Dynamic Web Project project named rdb-to-onto.

4 - Add the project to Tomcat 7.0.37 configured with Java 8.

5 - No MySQL, criar o banco rdbtoonto.

6 - Run the file rdb-to-onto\WebContent\db\rdb-to-onto.sql or according Appendix E, in MySQL to create the tables.

7 - Initialize Tomcat and access the address:

```
http://localhost:8080/rdb-to-onto/RdbToOnto.do
```

Obs.: The system is configured to access the database with the credentials root/12345. If necessary, change the data in the file rdb\to\onto\src\resources\hibernate.cfg.xml

APPENDIX E

SCRIPT OF MAPPING SCHEMA

```
--
-- Table structure for table 't001_database'
--

DROP TABLE IF EXISTS 't001_database';

/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 't001_database' (
  'C001_DATABASE_ID' int(11) NOT NULL AUTO_INCREMENT,
  'C001_DATABASE_NAME' varchar(255) NOT NULL,
  PRIMARY KEY ('C001_DATABASE_ID')
) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table 't002_table'
--

DROP TABLE IF EXISTS 't002_table';

/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 't002_table' (
  'C002_TABLE_ID' int(11) NOT NULL AUTO_INCREMENT,
  'C002_TABLE_TYPE' varchar(255) NOT NULL,
```

```

    'C001_DATABASE_ID' int(11) NOT NULL,
    'C002_PHYSICAL_NAME' varchar(255) NOT NULL,
    'C002_LOGICAL_NAME' varchar(255) NOT NULL,
    'C002_IND_ASSOCIATIVE' tinyint(1) NOT NULL DEFAULT '0',
    PRIMARY KEY ('C002_TABLE_ID'),
    KEY 'C001_DATABASE_ID_idx' ('C001_DATABASE_ID'),
    CONSTRAINT 'FK_T001_TO_T002' FOREIGN KEY ('C001_DATABASE_ID')
    REFERENCES 't001_database' ('C001_DATABASE_ID')
    ON DELETE NO ACTION ON UPDATE NO ACTION,
    CONSTRAINT 'FKt5v2hc1iahvdxl20y08a3j7hu' FOREIGN KEY ('C001_DATABASE_ID')
    REFERENCES 't001_database' ('C001_DATABASE_ID')
) ENGINE=InnoDB AUTO_INCREMENT=195 DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table 't003_column'
--

DROP TABLE IF EXISTS 't003_column';
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 't003_column' (
    'C003_COLUMN_ID' int(11) NOT NULL AUTO_INCREMENT,
    'C003_PHYSICAL_NAME' varchar(255) NOT NULL,
    'C003_LOGICAL_NAME' varchar(255) NOT NULL,
    'C002_TABLE_ID' int(11) NOT NULL,
    'C003_IND_PRIMARY_KEY' tinyint(1) NOT NULL DEFAULT '0',
    'C003_IND_FOREIGN_KEY' tinyint(1) NOT NULL DEFAULT '0',
    'C002_FK_TABLE_ID' int(11) DEFAULT NULL,

```

```

'C005_DATATYPE_ID' int(11) NOT NULL,
'C003_IND_DESCRIPTION' tinyint(1) DEFAULT NULL,
'C003_IND_ASSOCIATIVE_KEY' tinyint(1) DEFAULT NULL,
'C002_AK_TABLE_ID_1' int(11) DEFAULT NULL,
'C003_AK_COLUMN_ID_1' int(11) DEFAULT NULL,
'C002_AK_TABLE_ID_N' int(11) DEFAULT NULL,
'C003_AK_COLUMN_N' int(11) DEFAULT NULL,
'C003_LOGICAL_NAME_2' varchar(255) DEFAULT NULL,
'C003_IND_COLUMN_CHECK' tinyint(1) DEFAULT NULL,
'C003_IND_UNIQUE_KEY' tinyint(1) DEFAULT NULL,
PRIMARY KEY ('C003_COLUMN_ID'),
KEY 'FK_T002_TO_T003_idx' ('C002_TABLE_ID'),
KEY 'FK_T002_TO_T003_FK_ID_idx' ('C002_FK_TABLE_ID'),
KEY 'FK_T005_TO_T003_idx' ('C005_DATATYPE_ID'),
KEY 'FKro2eoxoxnx9sodk0lqmd7p9ys' ('C002_AK_TABLE_ID_1'),
KEY 'FKnucnnor18i92n364i49e9mwjy' ('C002_AK_TABLE_ID_N'),
CONSTRAINT 'FK43t3s791t1x8b07x0p49retvw' FOREIGN KEY ('C002_FK_TABLE_ID')
REFERENCES 't002_table' ('C002_TABLE_ID'),
CONSTRAINT 'FK99dqxjxkjrtwr6kch4hqkxsb8' FOREIGN KEY ('C005_DATATYPE_ID')
REFERENCES 't005_datatype_db' ('C005_DATATYPE_ID'),
CONSTRAINT 'FK_T002_TO_T003' FOREIGN KEY ('C002_TABLE_ID')
REFERENCES 't002_table' ('C002_TABLE_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT 'FK_T002_TO_T003_FK_ID' FOREIGN KEY ('C002_FK_TABLE_ID')
REFERENCES 't002_table' ('C002_TABLE_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT 'FK_T005_TO_T003' FOREIGN KEY ('C005_DATATYPE_ID')
REFERENCES 't005_datatype_db' ('C005_DATATYPE_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION,

```

```

CONSTRAINT 'FKbrq9bix9aktmakiqzk8gv6nb6' FOREIGN KEY ('C002_TABLE_ID')
REFERENCES 't002_table' ('C002_TABLE_ID'),
CONSTRAINT 'FKnucnnor18i92n364i49e9mwjy'
FOREIGN KEY ('C002_AK_TABLE_ID_N')
REFERENCES 't002_table' ('C002_TABLE_ID'),
CONSTRAINT 'FKro2eoxoxnx9sodk0lqmd7p9ys'
FOREIGN KEY ('C002_AK_TABLE_ID_1')
REFERENCES 't002_table' ('C002_TABLE_ID')
) ENGINE=InnoDB AUTO_INCREMENT=2505 DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table 't004_record'
--

DROP TABLE IF EXISTS 't004_record';
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 't004_record' (
  'C004_RECORD_ID' int(11) NOT NULL AUTO_INCREMENT,
  'C002_TABLE_ID' int(11) NOT NULL,
  'C004_TABLE_COLUMNS' varchar(255) NOT NULL,
  'C004_COLUMN_VALUES' varchar(255) DEFAULT NULL,
  PRIMARY KEY ('C004_RECORD_ID'),
  KEY 'FK_T004_TO_T002_idx' ('C002_TABLE_ID'),
  CONSTRAINT 'FK_T002_TO_T004' FOREIGN KEY ('C002_TABLE_ID')
REFERENCES 't002_table' ('C002_TABLE_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT 'FKi2vp9scqhet6tg6bdix1rm3v' FOREIGN KEY ('C002_TABLE_ID')

```

```

REFERENCES 't002_table' ('C002_TABLE_ID')
) ENGINE=InnoDB AUTO_INCREMENT=3139 DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table 't005_datatype_db'
--

DROP TABLE IF EXISTS 't005_datatype_db';
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 't005_datatype_db' (
  'C005_DATATYPE_ID' int(11) NOT NULL AUTO_INCREMENT,
  'C005_DESCRIPTION' varchar(255) NOT NULL,
  PRIMARY KEY ('C005_DATATYPE_ID')
) ENGINE=InnoDB AUTO_INCREMENT=51 DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table 't006_check_value'
--

DROP TABLE IF EXISTS 't006_check_value';
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 't006_check_value' (
  'C006_CHECK_VALUE_ID' int(11) NOT NULL AUTO_INCREMENT,
  'C006_DESCRIPTION' varchar(255) NOT NULL,
  'C006_ABBREVIATION' varchar(255) DEFAULT NULL,

```



```

        'C007_CHECK_SUBJECT_ID' int(11) NOT NULL,
PRIMARY KEY ('C006_CHECK_VALUE_ID'),
KEY 'FK_T007_TO_T006_idx' ('C007_CHECK_SUBJECT_ID'),
CONSTRAINT 'FK_T007_TO_T006' FOREIGN KEY ('C007_CHECK_SUBJECT_ID')
REFERENCES 't007_check_subject' ('C007_CHECK_SUBJECT_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT 'FKrdir765o3xw1u19373k8s0w19'
FOREIGN KEY ('C007_CHECK_SUBJECT_ID')
REFERENCES 't007_check_subject' ('C007_CHECK_SUBJECT_ID')
) ENGINE=InnoDB AUTO_INCREMENT=316 DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table 't007_check_subject'
--

DROP TABLE IF EXISTS 't007_check_subject';
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 't007_check_subject' (
        'C007_CHECK_SUBJECT_ID' int(11) NOT NULL AUTO_INCREMENT,
        'C007_DESCRIPTION' varchar(255) NOT NULL,
PRIMARY KEY ('C007_CHECK_SUBJECT_ID')
) ENGINE=InnoDB AUTO_INCREMENT=49 DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table 't008_database_domain'
--

```

```

DROP TABLE IF EXISTS 't008_database_domain';

/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;

CREATE TABLE 't008_database_domain' (
  'C008_DATABASE_DOMAIN_ID' int(11) NOT NULL AUTO_INCREMENT,
  'C008_DESCRIPTION' varchar(255) NOT NULL,
  'C001_DATABASE_ID' int(11) NOT NULL,
  PRIMARY KEY ('C008_DATABASE_DOMAIN_ID'),
  KEY 'FK_T001_TO_T008_idx' ('C001_DATABASE_ID'),
  CONSTRAINT 'FK_T001_TO_T008' FOREIGN KEY ('C001_DATABASE_ID')
    REFERENCES 't001_database' ('C001_DATABASE_ID')
    ON DELETE NO ACTION ON UPDATE NO ACTION,
  CONSTRAINT 'FKp2xywmq1bw74he5e61l0tyi9s'
    FOREIGN KEY ('C001_DATABASE_ID')
    REFERENCES 't001_database' ('C001_DATABASE_ID')
) ENGINE=InnoDB AUTO_INCREMENT=179 DEFAULT CHARSET=utf8;

/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table 't009_column_check_value'
--

DROP TABLE IF EXISTS 't009_column_check_value';

/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;

CREATE TABLE 't009_column_check_value' (
  'C003_COLUMN_ID' int(11) NOT NULL,
  'C006_CHECK_VALUE_ID' int(11) NOT NULL,

```

```

PRIMARY KEY ('C003_COLUMN_ID','C006_CHECK_VALUE_ID'),
KEY 'FK_T006_TO_T009_idx' ('C006_CHECK_VALUE_ID'),
CONSTRAINT 'FK_T003_TO_T009' FOREIGN KEY ('C003_COLUMN_ID')
REFERENCES 't003_column' ('C003_COLUMN_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT 'FK_T006_TO_T009' FOREIGN KEY ('C006_CHECK_VALUE_ID')
REFERENCES 't006_check_value' ('C006_CHECK_VALUE_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT 'FKe60pjh7s6347o2pvo7eyunmwi' FOREIGN KEY ('C006_CHECK_VALUE_ID')
REFERENCES 't006_check_value' ('C006_CHECK_VALUE_ID'),
CONSTRAINT 'FKiqha46f5t341cypf8uewk71c9' FOREIGN KEY ('C003_COLUMN_ID')
REFERENCES 't003_column' ('C003_COLUMN_ID')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table 't010_table_db_domain'
--

DROP TABLE IF EXISTS 't010_table_db_domain';
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 't010_table_db_domain' (
  'C008_DATABASE_DOMAIN_ID' int(11) NOT NULL,
  'C002_TABLE_ID' int(11) NOT NULL,
  PRIMARY KEY ('C008_DATABASE_DOMAIN_ID','C002_TABLE_ID'),
  KEY 'FK_T002_TO_T010_idx' ('C002_TABLE_ID'),
  CONSTRAINT 'FK1cftrdckmllvlye6vkj0qcja1'
  FOREIGN KEY ('C008_DATABASE_DOMAIN_ID')

```

```

REFERENCES 't008_database_domain' ('C008_DATABASE_DOMAIN_ID'),
CONSTRAINT 'FK_T002_TO_T010' FOREIGN KEY ('C002_TABLE_ID')
REFERENCES 't002_table' ('C002_TABLE_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT 'FK_T008_TO_T010' FOREIGN KEY ('C008_DATABASE_DOMAIN_ID')
REFERENCES 't008_database_domain' ('C008_DATABASE_DOMAIN_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT 'FKbq22p96ii7piowfutxf3tx3xq' FOREIGN KEY ('C002_TABLE_ID')
REFERENCES 't002_table' ('C002_TABLE_ID')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table 't011_class'
--

DROP TABLE IF EXISTS 't011_class';
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 't011_class' (
  'C011_CLASS_ID' int(11) NOT NULL AUTO_INCREMENT,
  'C011_CLASS_NAME' varchar(255) DEFAULT NULL,
  'C008_DATABASE_DOMAIN_ID' int(11) DEFAULT NULL,
  'C003_COLUMN_ID' int(11) DEFAULT NULL,
  'C016_ONTOLOGY_ID' int(11) NOT NULL,
  'C002_TABLE_ID' int(11) DEFAULT NULL,
  'C007_CHECK_SUBJECT_ID' int(11) DEFAULT NULL,
  'C004_RECORD_ID' int(11) DEFAULT NULL,
  PRIMARY KEY ('C011_CLASS_ID'),

```

```

KEY 'fk_T011_CLASS_T016_ONTOLOGY1_idx' ('C016_ONTOLOGY_ID'),
KEY 'fk_T011_CLASS_T002_TABLE1_idx' ('C002_TABLE_ID'),
KEY 'fk_T011_CLASS_T007_CHECK_SUBJECT1_idx' ('C007_CHECK_SUBJECT_ID'),
KEY 'fk_T011_CLASS_T004_RECORD1_idx' ('C004_RECORD_ID'),
KEY 'FKn2lohk5sq13mirac040oxygwy' ('C003_COLUMN_ID'),
KEY 'FKpl6kkwe55plwpe8o8lisup30t' ('C008_DATABASE_DOMAIN_ID'),
CONSTRAINT 'FKhk1j7oi6kv6ux38vudwb1xana' FOREIGN KEY ('C016_ONTOLOGY_ID')
REFERENCES 't016_ontology' ('C016_ONTOLOGY_ID'),
CONSTRAINT 'FKiay72pvdbcnrjj72blw09tqv' FOREIGN KEY ('C004_RECORD_ID')
REFERENCES 't004_record' ('C004_RECORD_ID'),
CONSTRAINT 'FKmb4fu278pd5pb7q4wmpkadudy'
FOREIGN KEY ('C007_CHECK_SUBJECT_ID')
REFERENCES 't007_check_subject' ('C007_CHECK_SUBJECT_ID'),
CONSTRAINT 'FKn2lohk5sq13mirac040oxygwy'
FOREIGN KEY ('C003_COLUMN_ID')
REFERENCES 't003_column' ('C003_COLUMN_ID'),
CONSTRAINT 'FKpl6kkwe55plwpe8o8lisup30t' FOREIGN KEY ('C008_DATABASE_DOMAIN_ID')
REFERENCES 't008_database_domain' ('C008_DATABASE_DOMAIN_ID'),
CONSTRAINT 'FKtjjkwwkrbksrm9mgpufg5yjf2' FOREIGN KEY ('C002_TABLE_ID')
REFERENCES 't002_table' ('C002_TABLE_ID'),
CONSTRAINT 'fk_T011_CLASS_T002_TABLE1' FOREIGN KEY ('C002_TABLE_ID')
REFERENCES 't002_table' ('C002_TABLE_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT 'fk_T011_CLASS_T004_RECORD1' FOREIGN KEY ('C004_RECORD_ID')
REFERENCES 't004_record' ('C004_RECORD_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT 'fk_T011_CLASS_T007_CHECK_SUBJECT1'
FOREIGN KEY ('C007_CHECK_SUBJECT_ID')
REFERENCES 't007_check_subject' ('C007_CHECK_SUBJECT_ID')

```

```

ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT 'fk_T011_CLASS_T016_ONTOLOGY1'
FOREIGN KEY ('C016_ONTOLOGY_ID')
REFERENCES 't016_ontology' ('C016_ONTOLOGY_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB AUTO_INCREMENT=291 DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table 't012_hierarchy'
--

DROP TABLE IF EXISTS 't012_hierarchy';
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 't012_hierarchy' (
  'C011_SUPER_CLASS_ID' int(11) NOT NULL,
  'C011_SUB_CLASS_ID' int(11) NOT NULL,
  PRIMARY KEY ('C011_SUPER_CLASS_ID','C011_SUB_CLASS_ID'),
  KEY 'fk_T012_HIERARCHY_T011_CLASS1_idx' ('C011_SUB_CLASS_ID'),
  CONSTRAINT 'FK76qbmccup4w1xtk0403g0tdph' FOREIGN KEY ('C011_SUPER_CLASS_ID')
REFERENCES 't011_class' ('C011_CLASS_ID'),
  CONSTRAINT 'FKcrh0tw9q6gxyheebua08qx1ud' FOREIGN KEY ('C011_SUB_CLASS_ID')
REFERENCES 't011_class' ('C011_CLASS_ID'),
  CONSTRAINT 'fk_T012_HIERARCHY_T011_CLASS1' FOREIGN KEY ('C011_SUB_CLASS_ID')
REFERENCES 't011_class' ('C011_CLASS_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

```

--

-- Table structure for table 't013_datatype_property'

--

```

DROP TABLE IF EXISTS 't013_datatype_property';

/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;

CREATE TABLE 't013_datatype_property' (
  'C013_DATATYPE_PROPERTY_ID' int(11) NOT NULL AUTO_INCREMENT,
  'C013_DATATYPE_PROPERTY_DESCRIPTION' varchar(100) DEFAULT NULL,
  'C016_ONTOLOGY_ID' int(11) DEFAULT NULL,
  'C018_DATATYPE_ID' int(11) DEFAULT NULL COMMENT '\n\n\n',
  'C013_IND_COMMON_CONCEPT' tinyint(1) DEFAULT NULL,
  'C013_IND_DESCRIPTION' tinyint(1) DEFAULT NULL,
  PRIMARY KEY ('C013_DATATYPE_PROPERTY_ID'),
  KEY 'fk_T013_DATATYPE_PROPERTY_T016_ONTOLOGY1_idx' ('C016_ONTOLOGY_ID'),
  KEY 'fk_T013_DATATYPE_PROPERTY_T018_DATATYPE_ONTO1_idx' ('C018_DATATYPE_ID'),
  CONSTRAINT 'FK67goxyjcnbtg7v99454g96hb6' FOREIGN KEY ('C018_DATATYPE_ID')
REFERENCES 't018_datatype_onto' ('C018_DATATYPE_ID'),
  CONSTRAINT 'FKsyftv96wq3p7g3eu1ljb9su8i'
FOREIGN KEY ('C016_ONTOLOGY_ID')
REFERENCES 't016_ontology' ('C016_ONTOLOGY_ID'),
  CONSTRAINT 'fk_T013_DATATYPE_PROPERTY_T016_ONTOLOGY1'
FOREIGN KEY ('C016_ONTOLOGY_ID')
REFERENCES 't016_ontology' ('C016_ONTOLOGY_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION,
  CONSTRAINT 'fk_T013_DATATYPE_PROPERTY_T018_DATATYPE_ONTO1'
FOREIGN KEY ('C018_DATATYPE_ID')

```

```

REFERENCES 't018_datatype_onto' ('C018_DATATYPE_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB AUTO_INCREMENT=414 DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table 't014_datatype_property_domain'
--

DROP TABLE IF EXISTS 't014_datatype_property_domain';
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 't014_datatype_property_domain' (
  'C013_DATATYPE_PROPERTY_ID' int(11) NOT NULL,
  'C011_CLASS_ID_DOMAIN' int(11) DEFAULT NULL,
  'C014_DATATYPE_SUBPROPERTY_OF' tinyint(1) DEFAULT NULL,
  KEY 'fk_T014_DATATYPE_PROPERTY_DOMAIN_RANGE_T011_CLASS1_idx'
    ('C011_CLASS_ID_DOMAIN'),
  KEY 'fk_T014_DATATYPE_PROPERTY_DOMAIN_RANGE_T013_DATATYPE_PROPER_idx'
    ('C013_DATATYPE_PROPERTY_ID'),
  CONSTRAINT 'FK9oelemvo01c3yg80xba9h4yj'
    FOREIGN KEY ('C013_DATATYPE_PROPERTY_ID')
      REFERENCES 't013_datatype_property' ('C013_DATATYPE_PROPERTY_ID'),
  CONSTRAINT 'FKcwn5tokbgf9tfrrsqclovbhif'
    FOREIGN KEY ('C011_CLASS_ID_DOMAIN')
      REFERENCES 't011_class' ('C011_CLASS_ID'),
  CONSTRAINT 'fk_T014_DATATYPE_PROPERTY_DOMAIN_RANGE_T011_CLASS1'
    FOREIGN KEY ('C011_CLASS_ID_DOMAIN')
      REFERENCES 't011_class' ('C011_CLASS_ID')

```



```

ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT 'fk_T014_DATATYPE_PROPERTY_DOMAIN_RANGE_T013_DATATYPE_PROPERTY1'
FOREIGN KEY ('C013_DATATYPE_PROPERTY_ID')
REFERENCES 't013_datatype_property' ('C013_DATATYPE_PROPERTY_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table 't015_instance'
--

DROP TABLE IF EXISTS 't015_instance';
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 't015_instance' (
  'C015_INSTANCE_ID' int(11) NOT NULL AUTO_INCREMENT,
  'C015_INSTANCE_DESCRIPTION' varchar(250) DEFAULT NULL,
  'C011_CLASS_ID' int(11) DEFAULT NULL,
  'C016_ONTOLOGY_ID' int(11) DEFAULT NULL,
  'C004_RECORD_ID' int(11) DEFAULT NULL,
  PRIMARY KEY ('C015_INSTANCE_ID'),
  KEY 'fk_T015_INSTANCE_T011_CLASS1_idx' ('C011_CLASS_ID'),
  KEY 'fk_T015_INSTANCE_T016_ONTOLOGY1_idx' ('C016_ONTOLOGY_ID'),
  KEY 'fk_T015_INSTANCE_T004_RECORD1_idx' ('C004_RECORD_ID'),
  CONSTRAINT 'FK1h5ys6tqg9hralvbjwxyx7aga' FOREIGN KEY ('C016_ONTOLOGY_ID')
REFERENCES 't016_ontology' ('C016_ONTOLOGY_ID'),
  CONSTRAINT 'FKa6r70f08p1oknil36fjflfsrv' FOREIGN KEY ('C011_CLASS_ID')
REFERENCES 't011_class' ('C011_CLASS_ID'),

```

```

CONSTRAINT 'FKom0jr9b5x26lmtkh5xv9gps3v' FOREIGN KEY ('C004_RECORD_ID')
REFERENCES 't004_record' ('C004_RECORD_ID'),
CONSTRAINT 'fk_T015_INSTANCE_T004_RECORD1' FOREIGN KEY ('C004_RECORD_ID')
REFERENCES 't004_record' ('C004_RECORD_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT 'fk_T015_INSTANCE_T011_CLASS1' FOREIGN KEY ('C011_CLASS_ID')
REFERENCES 't011_class' ('C011_CLASS_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT 'fk_T015_INSTANCE_T016_ONTOLOGY1'
FOREIGN KEY ('C016_ONTOLOGY_ID')
REFERENCES 't016_ontology' ('C016_ONTOLOGY_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB AUTO_INCREMENT=3377 DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table 't016_ontology'
--

DROP TABLE IF EXISTS 't016_ontology';
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 't016_ontology' (
  'C016_ONTOLOGY_ID' int(11) NOT NULL AUTO_INCREMENT,
  'C016_ONTOLOGY_NAME' varchar(255) DEFAULT NULL,
  'C001_DATABASE_ID' int(11) NOT NULL,
  PRIMARY KEY ('C016_ONTOLOGY_ID'),
  KEY 'fk_T016_ONTOLOGY_T001_DATABASE1_idx' ('C001_DATABASE_ID'),
  CONSTRAINT 'FKk22h5c31u6eve7r16o08ovn2r' FOREIGN KEY ('C001_DATABASE_ID')

```

```

REFERENCES 't001_database' ('C001_DATABASE_ID'),
CONSTRAINT 'fk_T016_ONTOLOGY_T001_DATABASE1'
FOREIGN KEY ('C001_DATABASE_ID')
REFERENCES 't001_database' ('C001_DATABASE_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table 't017_disjoint_class'
--

DROP TABLE IF EXISTS 't017_disjoint_class';
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 't017_disjoint_class' (
  'C011_CLASS_ID' int(11) NOT NULL,
  'C011_DISJOINT_CLASS_ID' int(11) DEFAULT NULL,
  PRIMARY KEY ('C011_CLASS_ID'),
  KEY 'fk_T017_CLASS_RESTRICTION_T011_CLASS1_idx' ('C011_CLASS_ID'),
  CONSTRAINT 'fk_T017_CLASS_RESTRICTION_T011_CLASS1'
  FOREIGN KEY ('C011_CLASS_ID')
  REFERENCES 't011_class' ('C011_CLASS_ID')
  ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table 't018_datatype_onto'

```

--

```

DROP TABLE IF EXISTS 't018_datatype_onto';

/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;

CREATE TABLE 't018_datatype_onto' (
  'C018_DATATYPE_ID' int(11) NOT NULL AUTO_INCREMENT,
  'C018_DESCRIPTION' varchar(255) NOT NULL,
  'C005_DATATYPE_DB_ID' int(11) NOT NULL,
  PRIMARY KEY ('C018_DATATYPE_ID'),
  KEY 'fk_T018_DATATYPE_ONTO_T005_DATATYPE_DB1_idx' ('C005_DATATYPE_DB_ID'),
  CONSTRAINT 'FKgc1g163ci8jv1lod3do7geu1v'
  FOREIGN KEY ('C005_DATATYPE_DB_ID')
  REFERENCES 't005_datatype_db' ('C005_DATATYPE_ID'),
  CONSTRAINT 'fk_T018_DATATYPE_ONTO_T005_DATATYPE_DB1'
  FOREIGN KEY ('C005_DATATYPE_DB_ID')
  REFERENCES 't005_datatype_db' ('C005_DATATYPE_ID')
  ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB AUTO_INCREMENT=51 DEFAULT CHARSET=utf8;

/*!40101 SET character_set_client = @saved_cs_client */;

```

--

-- Table structure for table 't019_object_property'

--

```

DROP TABLE IF EXISTS 't019_object_property';

/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;

CREATE TABLE 't019_object_property' (

```

```

'CO19_OBJECT_PROPERTY_ID' int(11) NOT NULL AUTO_INCREMENT,
'CO19_OBJECT_PROPERTY_DESCRIPTION' varchar(250) DEFAULT NULL,
'CO16_ONTOLOGY_ID' int(11) NOT NULL,
'CO19_IND_INVERSE_FUNCTIONAL' tinyint(1) DEFAULT NULL,
'CO19_MIN_CARDINALITY' tinyint(1) DEFAULT NULL,
PRIMARY KEY ('CO19_OBJECT_PROPERTY_ID'),
KEY 'fk_T019_OBJECT_PROPERTY_T016_ONTOLOGY1_idx' ('CO16_ONTOLOGY_ID'),
CONSTRAINT 'FKrp5ougah0fnomgf5ut3efqai3' FOREIGN KEY ('CO16_ONTOLOGY_ID')
REFERENCES 't016_ontology' ('CO16_ONTOLOGY_ID'),
CONSTRAINT 'fk_T019_OBJECT_PROPERTY_T016_ONTOLOGY1'
FOREIGN KEY ('CO16_ONTOLOGY_ID')
REFERENCES 't016_ontology' ('CO16_ONTOLOGY_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB AUTO_INCREMENT=765 DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table 't020_column_to_datatype_property'
--

DROP TABLE IF EXISTS 't020_column_to_datatype_property';
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 't020_column_to_datatype_property' (
  'CO03_COLUMN_ID' int(11) NOT NULL,
  'CO13_DATATYPE_PROPERTY_ID' int(11) NOT NULL,
  PRIMARY KEY ('CO03_COLUMN_ID', 'CO13_DATATYPE_PROPERTY_ID'),
  KEY 'fk_T003_COLUMN_has_T013_DATATYPE_PROPERTY_T013_DATATYPE_PRO_idx'
('CO13_DATATYPE_PROPERTY_ID'),

```

```

KEY 'fk_T003_COLUMN_has_T013_DATATYPE_PROPERTY_T003_COLUMN1_idx'
('C003_COLUMN_ID'),
CONSTRAINT 'FK1h8uwyrdjtr5py0sketus5q'
FOREIGN KEY ('C003_COLUMN_ID')
REFERENCES 't003_column' ('C003_COLUMN_ID'),
CONSTRAINT 'FK3fumgxcfi85ylh2xvw32th89f'
FOREIGN KEY ('C013_DATATYPE_PROPERTY_ID')
REFERENCES 't013_datatype_property' ('C013_DATATYPE_PROPERTY_ID'),
CONSTRAINT 'fk_T003_COLUMN_has_T013_DATATYPE_PROPERTY_T003_COLUMN1'
FOREIGN KEY ('C003_COLUMN_ID')
REFERENCES 't003_column' ('C003_COLUMN_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT 'fk_T003_COLUMN_has_T013_DATATYPE_PROPERTY_T013_DATATYPE_PROPE1'
FOREIGN KEY ('C013_DATATYPE_PROPERTY_ID')
REFERENCES 't013_datatype_property' ('C013_DATATYPE_PROPERTY_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table 't021_column_to_object_property'
--

DROP TABLE IF EXISTS 't021_column_to_object_property';

/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 't021_column_to_object_property' (
  'C003_COLUMN_ID' int(11) NOT NULL,
  'C019_OBJECT_PROPERTY_ID' int(11) NOT NULL,

```

```

PRIMARY KEY ('C003_COLUMN_ID','C019_OBJECT_PROPERTY_ID'),
KEY 'fk_T003_COLUMN_has_T019_OBJECT_PROPERTY_T019_OBJECT_PROPERTY_idx'
('C019_OBJECT_PROPERTY_ID'),
KEY 'fk_T003_COLUMN_has_T019_OBJECT_PROPERTY_T003_COLUMN1_idx'
('C003_COLUMN_ID'),
CONSTRAINT 'FKpb856xnpnnog7p41q1a3f99y5' FOREIGN KEY ('C003_COLUMN_ID')
REFERENCES 't003_column' ('C003_COLUMN_ID'),
CONSTRAINT 'FKt248lsd4i9j436pcbvu03ufc6'
FOREIGN KEY ('C019_OBJECT_PROPERTY_ID')
REFERENCES 't019_object_property' ('C019_OBJECT_PROPERTY_ID'),
CONSTRAINT 'fk_T003_COLUMN_has_T019_OBJECT_PROPERTY_T003_COLUMN1'
FOREIGN KEY ('C003_COLUMN_ID')
REFERENCES 't003_column' ('C003_COLUMN_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT 'fk_T003_COLUMN_has_T019_OBJECT_PROPERTY_T019_OBJECT_PROPERTY1'
FOREIGN KEY ('C019_OBJECT_PROPERTY_ID')
REFERENCES 't019_object_property' ('C019_OBJECT_PROPERTY_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table 't022_object_property_domain_range'
--

DROP TABLE IF EXISTS 't022_object_property_domain_range';
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 't022_object_property_domain_range' (

```

```

'CO19_OBJECT_PROPERTY_ID' int(11) NOT NULL,
'CO11_CLASS_ID_DOMAIN' int(11) DEFAULT NULL,
'CO11_CLASS_ID_RANGE' int(11) DEFAULT NULL,
KEY 'fk_T022_OBJECT_PROPERTY_DOMAIN_RANGE_T011_CLASS1_idx'
('CO11_CLASS_ID_DOMAIN'),
KEY 'fk_T022_OBJECT_PROPERTY_DOMAIN_RANGE_T011_CLASS2_idx'
('CO11_CLASS_ID_RANGE'),
KEY 'fk_T022_OBJECT_PROPERTY_DOMAIN_RANGE_T019_OBJECT_PROPERTY1_idx'
('CO19_OBJECT_PROPERTY_ID'),
CONSTRAINT 'FK7iil0daxusnbx6my8w3npl2j'
FOREIGN KEY ('CO19_OBJECT_PROPERTY_ID')
REFERENCES 't019_object_property' ('CO19_OBJECT_PROPERTY_ID'),
CONSTRAINT 'FKhlvs46m0wfrhwn54woa9hvsr2'
FOREIGN KEY ('CO11_CLASS_ID_DOMAIN')
REFERENCES 't011_class' ('CO11_CLASS_ID'),
CONSTRAINT 'FKpp8jc2psw73mtqyf2p41oycrf'
FOREIGN KEY ('CO11_CLASS_ID_RANGE')
REFERENCES 't011_class' ('CO11_CLASS_ID'),
CONSTRAINT 'fk_T022_OBJECT_PROPERTY_DOMAIN_RANGE_T011_CLASS1'
FOREIGN KEY ('CO11_CLASS_ID_DOMAIN')
REFERENCES 't011_class' ('CO11_CLASS_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT 'fk_T022_OBJECT_PROPERTY_DOMAIN_RANGE_T011_CLASS2'
FOREIGN KEY ('CO11_CLASS_ID_RANGE')
REFERENCES 't011_class' ('CO11_CLASS_ID')
ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT 'fk_T022_OBJECT_PROPERTY_DOMAIN_RANGE_T019_OBJECT_PROPERTY1'
FOREIGN KEY ('CO19_OBJECT_PROPERTY_ID')
REFERENCES 't019_object_property' ('CO19_OBJECT_PROPERTY_ID')

```



```
ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;
```

APPENDIX F

CASE STUDY - DETAILS OF DATABASE TABLES

Tables of case study	Used to
t008_ato	stores procedures for dental treatment.
t041_grupo_ato	Stores dental specialties.
t027_tipo_usuario_uniodonto	stores the type of patients.
t017_estado_civil	stores the types of marital status.
t305_grupo_sip	stores a group classification of procedures for the Product Information System of the Brazilian national health agency.
t002_pessoa	stores dentists data.
t006_plano	stores dental insurance plan.
t900_pais	store country data.
t901_estado	store state data.
t902_cidade	store city data.
t903_titulo	store a classification of streets.
t904_bairro	store neighborhood data.
t905_logradouro	store street data.
t005_cliente	store information of companies.
t003_associado	store information of clients.
t012_usuario_uniodonto	store information of patients.
t054_plano_usuario	store dental insurance plan of patients.
t026_form_cooperado	store data of dentist of a dental care.
t034_orcamento	store data of patient of a dental care.
t035_atos_orcamento	store data of procedure of a dental care.
t023_especialidade	store data of dental specialty.
t029_arcada	store data of dental arch.
t030_segmento	store data of dental segment.
T024_espec_coop	store the specialty of a dentist.
t009_plano_ato_padrao	store data of procedures covered by the dental insurance plan.