ENSMA : Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
LIAS : Laboratoire d'Informatique et d'Automatique pour les Systèmes

# THESE

pour l'obtention du Grade de

**DOCTEUR DE L'ÉCOLE NATIONALE SUPÉRIEURE DE MÉCANIQUE ET D'AÉROTECHNIQUE**

(Diplôme National — Arrêté du 7 août 2006)

Ecole Doctorale : Science et Ingénierie pour l'Information, Mathématiques
Secteur de Recherche : INFORMATIQUE ET APPLICATIONS

Présentée par :

**Youness BAZHAR**

**************************************************

# Handling Behavioral Semantics in Persistent Metamodeling Systems

# Extension des Systèmes de Métamodélisation Persistant avec la Sémantique Comportementale

**************************************************

Directeur de Thèse : **Yamine AIT-AMEUR**
Co-encadrant : **Stéphane JEAN**

**************************************************

Soutenue le 13 Décembre 2013
devant la Commission d'Examen

**<u>JURY</u>**

| | | |
|---|---|---|
| **Président :** | **Yves LEDRU** | Professeur, LIG / Université Joseph Fourier, Grenoble |
| **Rapporteurs :** | **Mohand-Said HACID** | Professeur, LIRIS / Université Claude Bernard Lyon 1 |
| | **Régine LALEAU** | Professeur, LACL / Université Paris-Est Créteil |
| **Examinateurs :** | **Ladjel BELLATRECHE** | Professeur, LIAS / ISAE-ENSMA, Chasseneuil du Poitou |
| | **Patrick FARAIL** | Ingénieur-Chef de projet, Airbus Operation SAS, Toulouse |
| | **Oscar PASTOR** | Professeur, Université polytechnique de Valence, Espagne |
| | **Yamine AIT-AMEUR** | Professeur, IRIT / INP-ENSEEIHT, Toulouse |
| | **Stéphane JEAN** | Maître de conférences, LIAS / Université de Poitiers |

# Merci à

**Yamine Aït-Ameur**, mon directeur de thèse, pour m'avoir guidé tout au long de cette thèse. Je le remercie pour son encadrement, pour l'esprit scientifique qu'il m'a transmis, et pour le temps et les efforts considérables qu'il a consacrés pour l'accomplissement de ce travail. Je le remercie également de m'avoir accueilli au LISI pendant la période de sa direction du laboratoire, et de m'avoir ouvert toujours les portes de son laboratoire actuel.

**Stéphane Jean**, mon co-encadrant de thèse, pour son encadrement, pour l'esprit critique qu'il m'a transmis, ses conseils précieux, ses retours d'expériences et aussi pour l'excellent travail qu'il a réalisé pendant sa thèse et sur lequel je me suis appuyé pour accomplir ce travail.

**Mohand-Said Hacid** et **Régine Laleau** pour m'avoir fait l'honneur d'être rapporteurs de cette thèse. Je leur exprime ma profonde reconnaissance d'avoir accepté cette lourde tâche.

**Ladjel Bellatreche**, **Patrick Farail**, **Yves Ledru**, **Oscar Pastor** pour avoir accepté de faire partie du jury de cette en tant qu'examinateurs. Je suis honoré pour l'intérêt qu'ils ont porté à ce travail.

**Mickaël Baron** pour son implication et son suivi des travaux menés dans le cadre de cette thèse. Je le remercie également pour son amitié et sa sympathie qui ont été pour moi une source de motivation.

**Frédéric Carreau** et **Claudine Rault** pour leur assistance administrative et logistique pendant toute la période de ma thèse.

Tous les membres du LIAS (**Yassine**, **Chedlia**, **Idir**, **Nabil**, **François**, **Amira**, **Ilyes**, **Laurent**, **Mikky**, **Ridou**, **Henri**, **Zoé**, **Brice**, **Selma**, **Georges**, **Thomas**, **Géraud**, **Ahcène**, **Loé**, **Linda**, **Manu**, **Pascal**, **Dilek**, **Bery**, **Mustapha**, **Allel**, **Christian**, **Ahmed**, **Valéry**, etc.) que j'ai côtoyé et avec qui j'ai passé des moments formidables et inoubliables.

**Mes parents**, **mes deux soeurs**, **mon frère**, les familles **Izague** et **Brett** qui ont toujours été d'un soutien continu et inconditionnel, et une source de motivation inépuisable.

# Contents

# Introduction

## Context

Since they appeared, databases have been continuously evolving. Relational databases were the most popular databases to store data. With the emergence of the object-oriented paradigm, both relational-object databases and object-oriented databases were introduced to support some object-oriented features such as the inheritance relationship. Then, the same pattern can be observed with the emergence of other types of databases such as XML databases, document-oriented databases or graph-oriented databases. Nowadays, the notion of model is used more than ever in software engineering and the produced models are more and more growing in size. Thus, a new generation of databases, called model repositories (also called model stores or metadata repositories), has emerged to store metamodels, models and their instances. Improving existing model repositories is the main goal of our thesis. Before presenting more precisely our objectives, let us first detail why the need of model repositories is becoming increasingly crucial.

The design of complex software systems are often based on different programming languages and platforms. In order to cope with this challenge, a software development methodology called Model-Driven Engineering (MDE) has been proposed. In this methodology, the design of complex systems use a set of heterogeneous models. These models represent a system from different viewpoints. For example, the modeling and analysis of embedded systems rely on different modeling languages such as SysML, UML/MARTE or AADL. The management of the models used to design a system is based on model management operations such as the generation of part of or of the complete software source code from models, or the transformation of a model in one modeling language into another model in another modeling language. By decoupling the system functionalities from the platform specific implementation, the goal of MDE is to improve the software development process and to ease software maintenance and reuse.

MDE has gained a lot of interest in industrial contexts thanks to the advantages it offers. MDE is widely used in various areas such as aeronautics and automotive and contributes to build multiple types of applications such as databases, domain specific languages (DSLs), Computer-Aided Software Engineering (CASE) tools and information systems. Moreover, MDE has been used to develop IT[1]

[1]Information Technology

solutions for systems integration and data exchange. The success of MDE has led to the development of several standards, tools, languages, etc. For instance, the Object Management Group (OMG) proposed the Model-Driven Architecture (MDA) [mda, 2003] as a standard specialization (i.e., implementation) of MDE. MDA is a software development methodology built around a set of standards (e.g., MOF [mof, 2011], UML [uml, 2011]), languages (e.g., ATL [Jouault and Kurtev, 2005], OCL [Cabot and Gogolla, 2012]) and tools (e.g., Eclipse Modeling Framework [emf, 2013a], Acceleo [acc, 2013]).

With the increasing usage of MDE, the management of very large models has appeared as one of the major challenge faced by this methodology. This challenge needs to be tackled when MDE has to be extensively used in industrial contexts [Clasen et al., 2012]. Indeed, a lot of domains such as e-commerce or engineering produce over-sized models and data. For instance, models may contain millions of elements (e.g., case of the reverse-engineering of big systems) and may describe large scale datasets (e.g., in genomics, the Uniprot[2] dataset gathers more than 200 GB of protein sequence resources). As most *MetaModeling Systems* (MMS) were initially designed for a management of models in main-memory, they fail to manage so large data and models.

To address the scalability problem of MDE, most of the proposed approaches aimed at improving the scalability of existing metamodeling and model management tools evolving in main memory. Different approaches arose such as:

- performing an incremental and partial management of large models [Jouault and Tisi, 2010]. The idea is to load in main memory, the required part of a large model only. Of course, this approach cannot be followed if an operation applies to the whole model;

- performing the management of large models in distributed settings [Clasen et al., 2012]. The idea consists in decomposing a management task into smaller tasks that can be performed on existing tools. This approach is limited to operations that can be decomposed into smaller and independent tasks;

- using a database as a back-end repository. This approach consists in equipping metamodeling systems with persistence solutions using object-relational mappings (e.g., Teneo [ten, 2013]) to store metamodels, models and instances in dedicated repositories called *model repositories* or *metadata repositories* (e.g., EMFStore [Koegel and Helming, 2010]). However, this approach has two main drawbacks: (i) repositories are only model warehouses since they are equipped with languages limited only to querying capabilities, so (ii) all model management tasks (transformation, code generation, etc.) require loading models in main memory in order to be processed.

Contrary to other paradigms (e.g., XML or the object-oriented paradigm), few work have been conducted to extend relational databases to manage models and metamodels as first-class entities. ConceptBase [Jarke et al., 2009b], OntoDB/OntoQL [Dehainsala et al., 2007, Jean et al., 2006a] and Rondo [Melnik et al., 2003] are examples of this approach which consists in defining systems for metamodeling and model management evolving completely in a database environment. These systems, called *Persistent MetaModeling Systems* (PMMSs), consist in (1) a model repository that stores metamodels, models and

[2]www.uniprot.org

instances while respecting the separation between the different metadata layers and preserving the conformity at the different abstraction levels, and (2) an associated exploitation language capable to create and manipulate the different model layers. PMMSs have been proposed to:

- get benefit from database scalability by exploiting the considerable amount of work that goes into data organization and query optimization in DBMS;
- offer a common repository for sharing models and data since DBMS provide the mechanisms to secure data access to the shared models;
- provide a declarative language allowing users to define models conforming to various metamodels.

Yet, if existing PMMSs exploit the characteristics of DBMS not yet supported by classical MMS, the arising question concerns the ability of PMMSs to provide the same functional capabilities offered by classical MMS. Indeed, in the literature we find that existing PMMSs support mostly the definition and the storage of structural and descriptive semantics of models by providing constructors of (meta)classes, (meta)attributes, primitives for expressing inheritance and association relationships, etc. However, PMMSs provide limited capabilities to define functions and procedures (behavioral semantics) that could be useful to handle advanced model management tasks such as model transformation or code generation. Currently, existing PMMSs use either low-level procedural languages (e.g., PL/SQL) which do not support the manipulation of high-level concepts (e.g., classes) [Dehainsala et al., 2007], or provide a fixed set of hard-coded operators (e.g., Match, Merge, Compose [Melnik et al., 2003]) devoted to specific model management tasks. ConceptBase remains the most advanced PMMS since it supports user-defined functions with membership constraints and external implementations. But, one of the limitations of ConceptBase is that it imposes a frozen deductive language to implement functions and thus, it cannot integrate programs defined in other languages (e.g., Java) nor web services. Moreover, it requires restarting the server each time a new external function is introduced which limits the availability of the PMMS (cold start).

## Our proposal

The aim of our work is to integrate the benefits of classical MMS together with the advantages of DBMS. Our approach claims the extension of database perspective. Our idea is to define a multi-level DBMS to store metamodels, models and instances with an associated SQL-like declarative exploitation language which provides the capability to define, manipulate (query, update and delete) these different abstraction layers. The different model management operations (e.g., model transformation) should also be available in PMMSs using flexible procedural mechanisms such as external programs (e.g., Java, C++) or web services. Thus, our approach focuses on the capability to dynamically define operators that could be exploited by the PMMS exploitation language. In particular, such operators could be implemented with internal database mechanisms (e.g., triggers, stored procedures) or external programs stored outside the database (e.g., Java or C++ programs), or with web services. As a consequence of this extension, we will be able to perform advanced model management tasks such as model transformations in database, trigger web services from databases. Information exchange and data integration could be also supported in this context.

To reach this objective, our work proposes to use a persistent solution within database systems. More precisely, the solution we suggest is based on the OntoDB/OntoQL PMMS but it can be applied to any other PMMS. If the OntoDB/OntoQL PMMS supports the structural manipulation of models and metamodels through its associated language (OntoQL), the definition of the behavior of models elements is not yet supported. Consequently, this system is not complete (in the computational sense) and needs to be extended in order to support the expression of behavioral semantics (e.g., operations, constraints, expressions, derivations, etc.). Since this system supports the manipulation of models as first-class objects through its metamodeling capabilities, this extension will enrich the OntoDB/OntoQL PMMS and will support models processing in order to achieve operations such as model transformations, data integration, constraints checking, etc. In this thesis, our work contributes to:

- the definition of a set of requirements for a complete PMMS and a state of the art that shows that existing PMMSs do not fulfill them;
- the formal definition of a PMMS data model and an associated exploitation language which include the procedural concepts of models and metamodels;
- the implementation of our proposition (the *BeMoRe* PMMS) with some preliminary experiments on its scalability;
- the development of three complete use cases in different domains to show the validity and usefulness of our approach.

## Structure of the thesis

This thesis is structured as follows.

**Chapter 1** introduces the notion of Model-Driven Engineering (MDE) including the concepts of metamodeling and model management which are of particular interest in this thesis.

**Chapter 2** presents existing persistence solutions dedicated to metamodeling and models management. We first describe the approaches used by classical MMS to overcome the scalability problem of MDE. Then, we introduce the notion of Persistent MetaModeling System (PMMS), on which we focus our work, and we discuss the capabilities and limitations of existing systems.

**Chapter 3** defines the requirements for a complete PMMS. These requirements integrate metamodeling and model management capabilities together with benefits of database systems (e.g., scalability, querying capability). Finally, we analyze the existing PMMSs according to the defined requirements.

**Chapter 4** exposes the formal extension of the PMMS metametamodel with new concepts to support the definition of model management operations that can be implemented using flexible mechanisms. Then, this chapter presents the extension of the logical metametamodel and metamodel schema of the PMMS repository. Finally, we introduce the formal extension of the algebra of the PMMS exploitation language with operators for the definition and the exploitation of operations and their associated implementations.

**Chapter 5** presents the extension of the OntoQL language with new instructions to define model and data management operations and implementations, and with the capability to invoke the defined operations in OntoQL statements. Besides, this chapter presents the technical aspects of the implementation of our approach, especially the execution process of OntoQL statements including operations invocations. Furthermore, this chapter proposes a small study of the scaling of our approach which is a part of the perspectives of our work.

**Chapter 6** presents a use case of our approach for managing non canonical concepts i.e., derived concepts in *Ontology-Based DataBases* (OBDBs) which are databases dedicated to the storage of ontologies.

**Chapter 7** presents the utilization of our proposition to improve an OBDB design methodology using the work achieved in Chapter 6.

**Chapter 8** addresses a use case of our approach which consists in using operations to perform model transformations and model analysis within PMMSs in the context of real-time systems design and analysis.

Finally, we list the conclusions of our proposition and we expose some future directions opened by our work.

# Part I

# State of the Art

Chapter 1

# Metamodeling and model management

## Contents

**Abstract.** Recently, Model-Driven Engineering (MDE) has been widely used in order to (i) build high quality software, (ii) improve the development process of software and (iii) facilitate their maintenance. With the emergence of MDE, several architectures, standards, tools and languages have been developed. In this chapter, we present the different notions of MDE relevant to our problem. These notions relate to the MOF architecture defined to manipulate models and metamodels as well as the model management operations that can be performed on them.

# 1 Introduction

During the recent years, model-driven engineering (MDE) has gained a lot of interest as it has the potential to speed up the software development process. MDE is applied in various engineering fields like aeronautics and automotive, and used to build different types of applications such as databases, domain specific languages (DSLs), CASE (Computer-Aided Software Engineering) tools and information systems. Moreover, MDE plays a central role in multiple IT solutions such as systems integration and data exchange. MDE includes various techniques devoted to the definition and the exploitation of metamodels and models. It involves two main activities: (1) metamodeling and modeling for defining models to describe the different aspects of systems, and (2) model management to represent the different models processings such as transformation, comparison, source code generation, archiving and model annotation.

The emergence of MDE has lead to the Model-Driven Architecture (MDA) [mda, 2003] which is a specialization of MDE. MDA has been proposed by the Object Management Group (OMG) and defined around many standards (e.g., UML [uml, 2011], MOF [mof, 2011], XMI [xmi, 2011]), languages (e.g., ATL [Jouault and Kurtev, 2005], Acceleo [acc, 2013], OCL [Cabot and Gogolla, 2012]) and tools (e.g., EMF [emf, 2013a] (Eclipse Modeling Framework), Kermeta [Jézéquel et al., 2009], Epsilon [Kolovos et al., 2013], Topcased [top, 2013]) that are dedicated to metamodeling and/or to accomplish specific model management tasks.

This chapter overviews metamodeling and model management techniques addressed in this thesis. Indeed, we start by introducing a background on MDE (Section 2). Then, we present the MOF metamodeling architecture (Section 3). Section 4 presents an overview of two model management activities: model transformation and code generation. Section 5 discusses specific issues related to metamodeling and model management and concludes this chapter.

## 2 Background on MDE

The main purposes of Model-Driven Engineering (MDE) are (i) to define models describing precisely real systems according to different viewpoints, and (2) to exploit the defined models in order to speed up the software development process. MDE relies on the notion of model which is a key element around which this discipline has been defined. Next section presents the notion of model and gives its different semantics.

### 2.1 The notion of model

The notion of model is a key element in MDE. A model depends on the viewpoint and the purpose. Several definitions of the notion of model exist in the literature. Misnky defines a model as follows: For an observer B, an object A* is a model of an object A to the extent that B can use A* to answer questions of interest about A [Minsky, 1968]. We adopt this definition in our work. Models are characterized by different semantics [Pierra et al., 1995]:

- *structural semantics*: one of the ways of understanding the real world is the formulation of categories (i.e., classes, types of entities) of objects (instances), their representation and differentiation. These categories are organized through relationships. A particularly important relationship is the generalization/specialization that goes from the general to the particular and defines hierarchies or, more generally acyclic graphs;

- *descriptive semantics*: associates properties (or attributes) to classes so that it differentiates between instances of classes;
- *behavioral semantics*: corresponds to the expression of objects behaviors and reasoning rules that can be applied to the different objects of each class. This type of semantics is represented by rules, functions and procedures.

Next subsection presents the vision of models and their classification according to the Model-Driven Architecture (MDA) approach.

## 2.2 The MDA approach

Model-Driven Architecture (MDA) [mda, 2003] is a software development methodology proposed by the Object Management Group (OMG). The purpose of MDA is to build high quality software based on models. Several standards have been defined in the context of the MDA approach. For instance, the Meta-Object Facility [mof, 2011] has been proposed by the OMG. It defines a four-layered metamodeling architecture gathering all the abstraction layers. UML (Unified Modeling Language) [uml, 2011] has been proposed as an object-oriented modeling language standard. XMI [xmi, 2011] defines an XML-based exchange format for UML models and instances. QVT [qvt, 2011] sets up the specifications for model transformation languages.

The objective of MDA is to separate the specification and the implementation of systems. In this context, MDA distinguishes three types of models illustrated in Figure 1.1 (the *Y* schema).

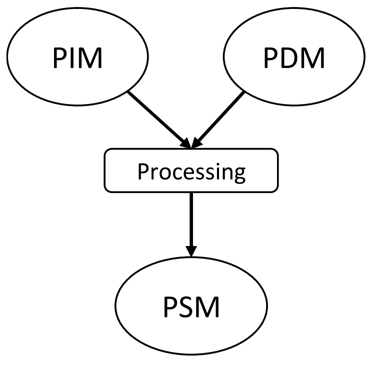


Figure 1.1: The *Y* schema of the MDA approach

- *Platform Independent Model* (PIM): represents the system independently of the platform that implements it.
- *Platform Dependent Model* (PDM): specifies the platform model of implementation (e.g., .NET, CORBA).

- *Platform Specific Model* (PSM): is the obtained executable model resulting from the transformation combining PIM and PDM.

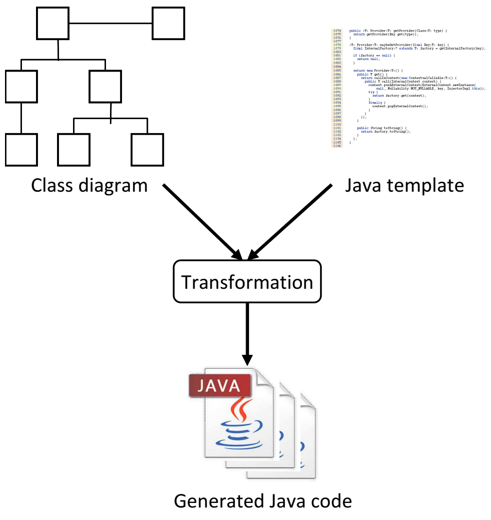


Figure 1.2: An instance of the *Y* schema

*Example.* Figure 1.2 shows an example illustrating the Y schema of the MDA approach. In this example, the PIM is represented by the UML class diagram while the PDM is the model that specifies mappings between the class diagram concepts and Java concepts i.e., the template that specifies how to generate the Java code from the class diagram. The PSM is the resulting generated Java code.

The definition of a model requires the knowledge of the metamodel that describes the modeling formalism used to design the model. The task of defining a metamodel is called *metamodeling*. Next section introduces the notion of metamodeling and presents in particular the MOF [mof, 2011] metamodeling architecture which is a standard of the OMG.

## 3 Metamodeling

Metamodeling is the task that consists in defining a model that describes precisely a modeling language. This model is called *metamodel*. In our work, we use extensively the notions of metamodel and metametamodel. These notions are presented in the next sections.

### 3.1 Definitions

#### 3.1.1 Metamodel

A metamodel is a model that describes a modeling formalism. It explicits the constructors, the relationships between concepts and the rules and the constraints that models (instances of metamodels) must satisfy. A model conforms to a metamodel as a program conforms to a grammar or as an XML file conforms to an XML schema. The definition of metamodel that we adopt in our work is the one proposed by Pierra and al. [Pierra et al., 1995].

**Definition 1**

***Metamodel:*** *is a set of precise notations, associated to a semantics and reasoning rules so that it is able to represent a model [Pierra et al., 1995].*

### 3.1.2 Metametamodel

A metamodel is designed using a language devoted to define modeling languages. This language is itself described through a model called *metametamodel*. A metametamodel is a model that has the ability to define metamodels and to describe itself (reflexivity).

**Definition 2**

***Metametamodel:*** *is a model that provides primitives to define and describe metamodels.*

Next section presents the MOF metamodeling architecture that defines different abstraction layers.

## 3.2 The MOF metamodeling architecture

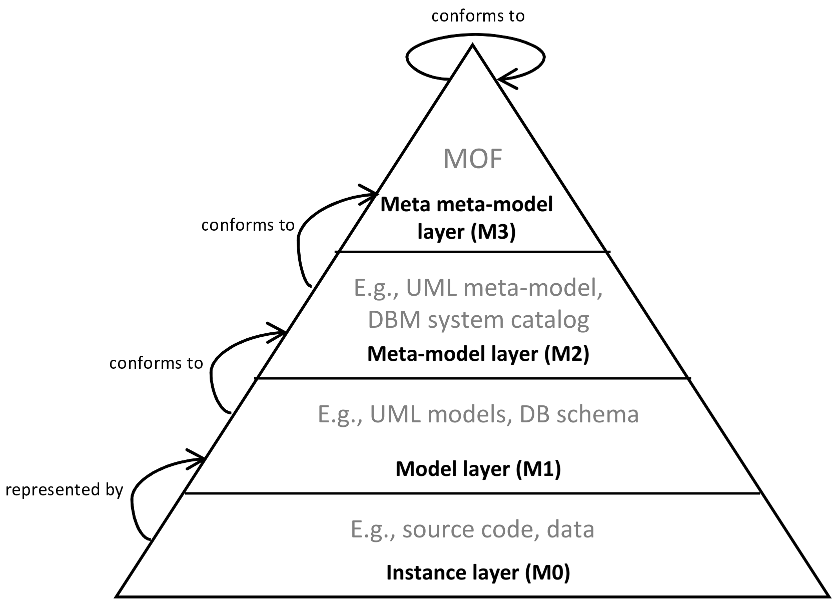


Figure 1.3: The MOF architecture

The Meta-Object Facility (MOF) [mof, 2011] is a metamodeling architecture standard proposed by the Object Management Group (OMG). It consists in four layers of metadata (Figure 1.3) corresponding to the different levels of abstraction: the instance layer (M0), the model layer (M1), the metamodel layer (M2) and the metametamodel layer (M3).

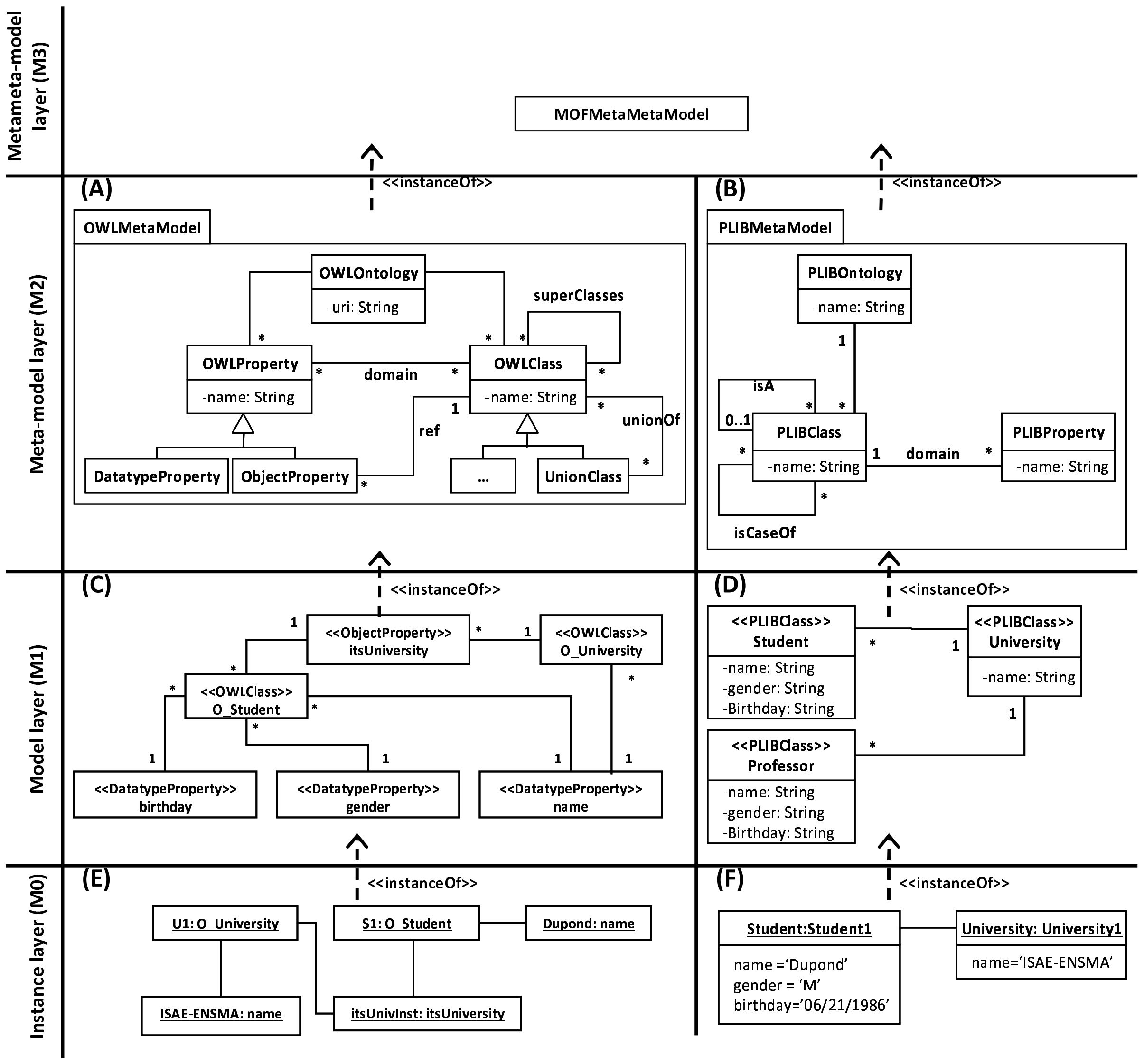


Figure 1.4: A running modeling example

Figure 1.4 gives a modeling example conforming to these four layers of the MOF architecture. Each of these layers is detailed below.

### 3.2.1 The metametamodel layer (M3)

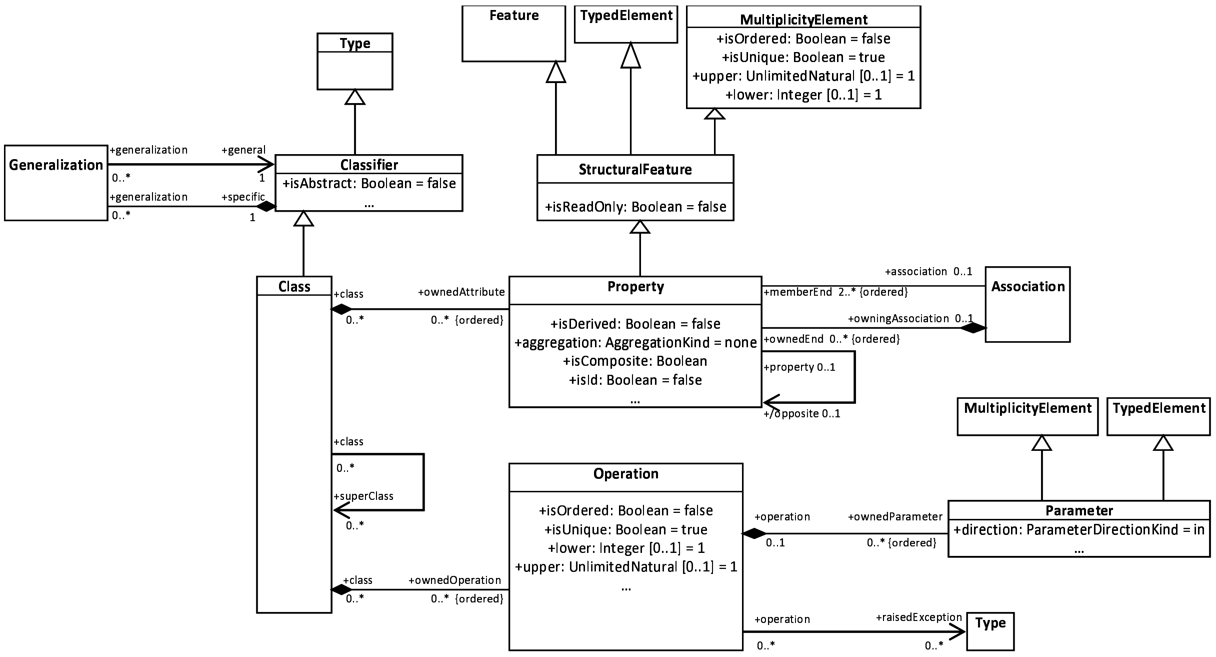


Figure 1.5: The MOF metametamodel

A metametamodel is the language facility used to define metamodels. The metametamodel layer is the last level of the MOF architecture. Indeed, this metametamodel is reflexive i.e., it describes itself. The metametamodel of the MOF architecture is presented in Figure 1.5.

The MOF metametamodel provides the essential concepts to define any modeling formalism (metamodel). In particular, it provides constructors for structural elements (e.g., the `Class`, `Classifier` and `Type` classes) organized in a hierarchy using the inheritance relationship. These structural elements are described by properties (the `Property` class). The concept of `Property` has different features. For instance, a property may be derived from one or many other properties (the `isDerived` attribute) and may be an identifier of the class it belongs to (the `isId` attribute). Moreover, a property may have an inverse property. The MOF metametamodel supports the definition of operations (the `Operation` class) i.e., functions and procedures with membership constraints. An operation may have ordered parameters (the `Parameter` class). Operations are useful for different tasks such as computing derived properties.

### 3.2.2 The metamodel layer (M2)

A metamodel is a conceptualization of a modeling language. The metamodel layer (M2) provides the structural description of modeling formalisms/languages and the rules that these languages shall respect. For instance, the descriptions of modeling formalisms like UML [uml, 2011], PLIB [Pierra and Sardet, 2010] or OWL [Dean and Schreiber, 2004] are defined at the metamodel layer.

The metamodel layer (M2) of the example in Figure 1.4 defines two metamodels: a part of a simplified metamodel of the Ontology Web Language (OWL) [Dean and Schreiber, 2004] (A) and a part of a simplified metamodel of the Parts LIBrary (PLIB) language (B) [Pierra and Sardet, 2010]. Both languages are used to design ontologies. In particular, OWL is dedicated to design ontologies for the Semantic Web whereas PLIB is devoted to design ontologies in engineering domains such as mechanics or aeronautics. PLIB and OWL possess different features. For instance, an OWL ontology is defined using classes and properties. An OWL property may be either a data type property (the `DatatypeProperty` class) i.e., a property whose range is a simple type (e.g., string, boolean), or an object property (the `ObjectProperty` class) referencing an OWL class instance. An OWL property may have several classes as domain and can be sub properties of several properties. Besides, OWL supports derived classes. For example, the notion of `UnionClass` represents an OWL class defined by the union of several OWL classes. A PLIB ontology is defined by classes and properties. Each property has a domain (the class it belongs to) and a range i.e., the type of the property. A simple inheritance relationship between classes is permitted in PLIB.

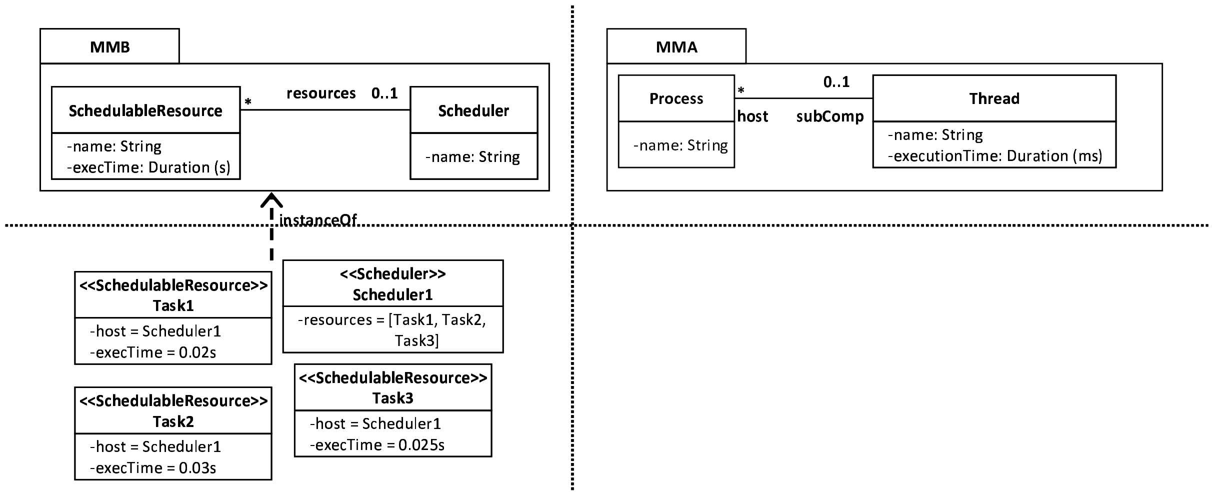


Figure 1.6: A metamodeling example

Figure 1.6 represents another metamodeling example . It shows two metamodels (`MMA` and `MMB`) and a model conforming to the `MMB` metamodel. Both metamodels can be used, for instance, to design real-time tasks that are processed by schedulers. The `MMB` metamodel defines a system as a set of schedulable resources (the `SchedulableResource` class) i.e., tasks, and schedulers (the `Scheduler` class) that process tasks. Each schedulable resource is characterized by a name and an execution time expressed in second, while each scheduler is characterized by a name. The `MMA` metamodel defines a system as a set of threads (the `Thread` class) i.e., tasks, and processes (the `Process` class). Each thread is characterized by a name and an execution time, expressed in millisecond, while each process is described by a name.

*Note.* The MOF architecture defines an extensible metamodel layer where several metamodels can be defined simultaneously .

### 3.2.3 The model layer (M1)

The model layer (M1) represents models. A model is expressed using a modeling language/formalism (metamodel) defined at the metamodel layer (M2). The model layer (M1) of the example in Figure 1.4 sketches the models (C) and (D) conforming respectively to the OWL metamodel and the PLIB metamodel. Both models represent the same ontology which defines 3 classes (`Student`, `Professor` and `University`) described by different properties. This ontology is expressed using the PLIB and the OWL formalisms.

The model of Figure 1.6 is an instance of the `MMB` metamodel. This model defines three schedulable resources (`Task1`, `Task2` and `Task3`) and a scheduler (`Scheduler1`).

### 3.2.4 The instance layer (M0)

The instance layer (M0) represents the real world. This level can support for example data, traces of a program execution, etc. Elements of the instance layer are instances of models (defined at the model layer (M1)). The instance layer (M0) of our example in Figure 1.4 defines two instances: (E) instance of the model (C), and (F) instance of the model (D).

The examples of Figure 1.4 and Figure 1.6 show the multiplicity of formalisms available to design and manage ontologies and real-time systems. Metamodels, models and instances of our examples can be designed using CASE tools such as Eclipse Modeling Framework (EMF) or starUML [sta, 2013], etc.

With the development of software engineering tools and techniques, models are subjects to multiple and different manipulations [Paige et al., 2011]. Indeed, models can be transformed (e.g., UML class diagram to DBMS schema), used for source code generation (e.g., generate the corresponding Java code of a class diagram), archived, compared, analyzed, etc. In the next section, we present an overview of two model management activities: model transformation and code generation since they are important model management activities.

## 4 Model management

The purpose of MDE is the exploitation of models once they are defined. In this section, we overview two model management activities which are model to model transformations and model to text transformations which correspond to code generation.

### 4.1 Model to model transformation

Model to model transformation or M2M (Model to Model) transformation consists in transforming, with a set of transformation rules, or a program, a source model ($M_s$), conforming to a source metamodel ($MM_s$), to a target model ($M_t$), conforming to a target metamodel ($MM_t$). Query View Transformation (QVT) [qvt, 2011] is the standard specification proposed by the OMG for model transformation languages. Two types of model transformation can be distinguished:

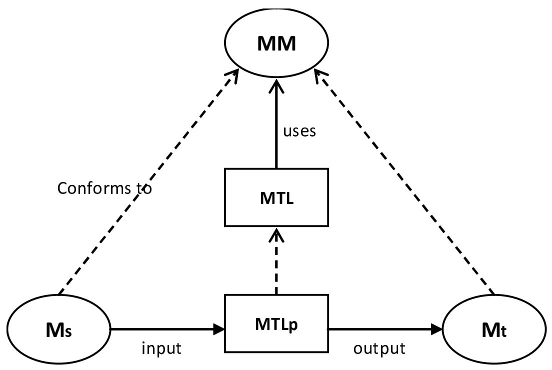


Figure 1.7: Endogenous model transformation

- *endogenous model transformation* (Figure 1.7): concerns the model transformation where the source and target models ($MM_s$ and $MM_t$) conform to the same metamodel ($MM$). Figure 1.7 illustrates this notion;
- *exogenous model transformation* (Figure 1.8): is a model transformation where the source and target models conform to distinct metamodels ($MM_s \neq MM_t$). Figure 1.8 conceptualizes the notion of exogenous model transformation.

$MTL$ and $MTL_p$ represent respectively the model transformation language and the model transformation program.

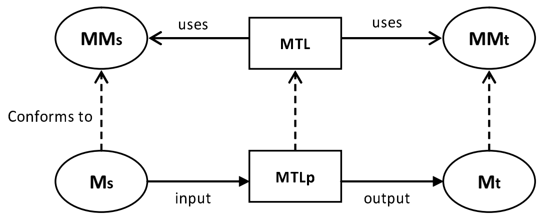


Figure 1.8: Exogenous model transformation

*Example.* Listing 1.1 presents an extract of a model transformation program that transforms PLIB models to OWL. This program is expressed with the ATL (Atlas Transformation Language) language [Jouault and Kurtev, 2005] which is a declarative and imperative M2M transformation language.

Listing 1.1: An example of an exogeneous model transformation program with ATL

```
module PLIB2OWL;

create OUT: OWL from IN: PLIB;

rule PLIBClass2OWLClass {
        from
                s: PLIB!PLIBClass
        to
                t: OWL!OWLClass (
                        name <- s.name,
                    comment <- s.remark
            )
}

rule PLIBProperty2OWLProperty {
    from
            s: PLIB!PLIBProperty
    to
            t: OWL!OWLProperty (
                    name <- s.name,
                    comment <- s.remark
            )
}
```

*Meaning.* This program transforms PLIB models to OWL ones as it takes as input a PLIB model and returns an OWL model as output. This program defines two rules. The first rule (`PLIBClass2OWLClass`) transforms a PLIB class to an OWL class. Indeed, the `name` attribute of a PLIB class is transformed to the `name` attribute of the OWL class while the `remark` attribute of PLIB class is transformed to the `comment` attribute of the OWL class. The second rule (`PLIBProperty2OWLProperty`) transforms a PLIB property to an OWL property in the same way as the first rule. Using this model transformation program, we can transform the PLIB model (D) of our example to the OWL model (C).

*Example.* Listing 1.2 presents an extract of the ATL transformation program of MMB models to MMA.

Listing 1.2: An example of an exogeneous model transformation program with ATL

```
module MMB2MMA;

create OUT: MMA from IN: MMB;

rule SchedulableResource2Thread {
    from
            s: MARTE!SchedulableResource
    to
            t: AADL!Thread (
                    name <- s.name,
                    executionTime <- s.execTime*1000
            )
}

rule Scheduler2Process {
    from
            s: MARTE!Scheduler
    to
            t: AADL!Process (
                    name <- s.name,
                    subComp <- s.resources
            )
}
```

*Meaning.* The first rule (`SchedulableResource2Thread`) transforms schedulable resources to threads. The name of a schedulable resource is transformed to the name of the corresponding thread. As the execution time of schedulable resources is expressed in second, the execution time of threads is multiplied by 1000 (the execution time of threads is expressed in millisecond). The second rule (`Scheduler2Process`) transforms a scheduler to a process. The resources of a scheduler are transformed to subcomponents of the process. The produced model is given in Figure 1.9.

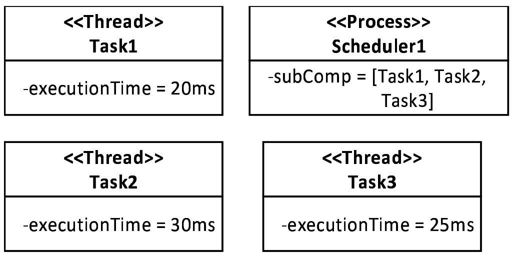


Figure 1.9: The resulting model of the model transformation `MMB2MMA`

Several tools and languages are dedicated to model-to-model transformations such as ATL [Jouault and Kurtev, 2005]. Besides, some metamodeling languages offering procedural capabilities (e.g., Kermeta [Jézéquel et al., 2009]) can be used to achieve model-to-model transformations. Next section presents the notion of model to text transformation.

## 4.2 Model to text transformation

Model to text transformation (M2T) consists in generating text from a source model ($M_s$), conforming to a source metamodel ($MM_s$), using a text generation template (program). Figure 1.10 illustrates this notion.

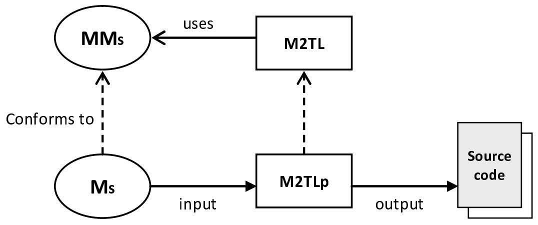


Figure 1.10: M2T transformation

Several languages have been proposed for code generation. For instance, Acceleo [acc, 2013] is a M2T transformation language devoted to define templates for code generation. Besides, Acceleo provides predefined code generation templates (e.g., to generate the Java code of UML class diagrams).

*Example.* Listing 1.3 presents a part of a model to text transformation program that generates the Java code of UML class diagrams.

Listing 1.3: An example of code generation template

```
[file (aEClass.name.concat('.java'), ...)]

public class [aEClass.name/] {

[for (aEAttribute : EAttribute | aEClass.eAllAttributes)]
    [aEAttribute.eType.instanceClassName/] [aEAttribute.name/];
[/for]
}
```

*Meaning.* This program creates a Java file corresponding to each class of an UML class diagram. Each generated Java class corresponds to a class of the model on which the generation is applied. The generated code defines Java classes with attributes.

The execution of the code generation template creates Java files containing the generated Java code corresponding to the classes of our model (Figure 1.4). For instance, the generated Java code of the Student class is given in Listing 1.4.

Listing 1.4: The generated Java code of the Student class

```
public class Student {

    String name;
    String gender;
    String birthday;

}
```

## 5 Conclusion

In this chapter we have introduced the notions of metamodeling and model management. Concerning metamodeling, we have seen that the MOF provides a four-layered metadata architecture that supports the different levels of abstraction. Moreover, models have different semantics (structural, descriptive and behavioral). Regarding model management, we have seen that models can be subjects to different manipulations like transformation and source code generation. However, existing metamodeling and model management tools and systems may show some limitations when they have to treat large scale and voluminous models. Indeed, the increasing size of models and their instances in industrial contexts raises the issue of scalability as one of the major challenges of MDE. Indeed, most of metamodeling and model management tools show limitations when they face large-scale models and data because large models and instances do not fit in main memory. Moreover, simple exchange files may not be sufficient for sharing large scale models and data. Likewise, industries need common platforms to share heterogeneous models and data. Thus, persistent solutions (based on databases) offering an efficient exploitation are required to store and manipulate oversized metamodels, models and data.

Next chapter is dedicated to the presentation of the existing solutions based on databases to store and manipulate models together with their instances. We focus on solutions based on databases as we follow this approach in our work.

Chapter 2

# Persistent solutions for metamodeling and model management

**Contents**

**Abstract.** In the previous chapter, we have explained that classical metamodeling and model management tools (evolving in central memory) show some insufficiencies related to

scalability. This issue is raised especially when classical metamodeling and model management systems face over sized models. One approach to solve this problem consists in optimizing existing main-memory metamodeling systems or associating them with databases. In this chapter we present the different database-based solutions that have been proposed in the literature to overcome the problem of scalability faced by classical metamodeling and model management tools.

## 1 Introduction

In the previous chapter we have presented the main notions of MDE that we use in this thesis. These notions encompass the MOF architecture defined for metamodeling and model management platforms as well as the main operations that are usually performed on metamodels and models (model transformation and code generation). As several studies pointed out the increasing adoption of MDE in industrial contexts, they highlight scalability as an important challenge [Clasen et al., 2012]. Indeed, existing metamodeling and model management tools evolving in main memory face scalability problems when they need to manage large models composed of a big amount of elements. These over-sized models are not unusual in practice. For example, this kind of models are produced when complex systems are defined (e.g., in aeronautics) or when available large datasets are reused (e.g., open data, social networks or scientific repositories).

Two main approaches have been followed to overcome the scalability issues raised by the management of large models. The first approach consists in improving the scalability of existing metamodeling and model management tools evolving in main memory using different mechanisms (e.g., incremental model management [Jouault and Tisi, 2010], model stores [Koegel and Helming, 2010]). The second approach that we followed in our work, consists in defining systems for metamodeling and model management evolving completely in a database environment (e.g., ConceptBase [Jarke et al., 2009b], OntoDB/OntoQL [Dehainsala et al., 2007, Jean et al., 2006a], Rondo [Melnik et al., 2003]). These systems, called *Persistent MetaModeling Systems*, consist in (1) a model repository that stores metamodels, models and instances while respecting the separation between the different metadata layers and preserving the conformity at the different abstraction levels, and (2) an associated exploitation language possessing the capability to create and manipulate the different model layers. Additionally, PMMSs can be used as common settings to share voluminous and heterogeneous models since simple exchange files may not always be sufficient. PMMSs have been proposed to:

- overcome issues related to scalability: database environments are able to store and manage over-sized models and instances;
- avoid problems related to the heterogeneity of metamodels and models: in software engineering, models are defined using different formalisms, and consequently a PMMS shall support different models whatever are the formalisms used to design them;
- offer a common repository for sharing models and data: DBMSs are enough mature to guarantee a secured sharing of models and data while respecting accessibility constraints.

Yet, if existing PMMSs overcome some issues faced by MMS, the question that arises concerns the capability of PMMSs to fulfill the functional capabilities offered by classical metamodeling and model management tools evolving in main memory. Indeed, existing PMMSs focus more on the structural and descriptive semantics, but they provide restricted capabilities to define procedural semantics. For instance, some of the existing PMMSs use either low-level procedural languages (e.g., PL/SQL) or provide hard-coded operators (e.g., Match, Compose [Melnik et al., 2003]) that are dedicated to specific model management activities.

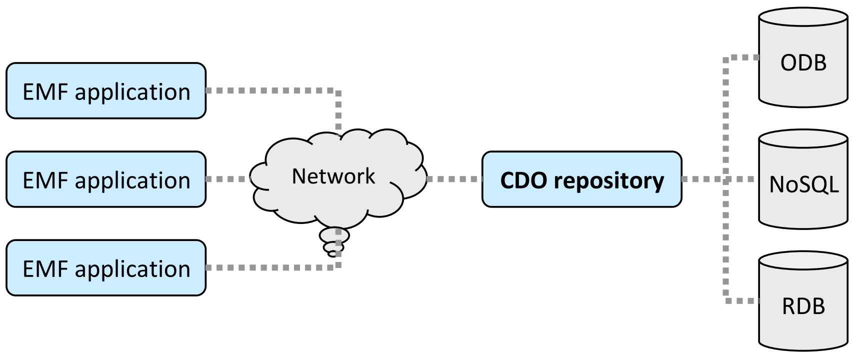


Figure 2.1: Architecture of the CDO model repository

This chapter introduces some of the existing model repositories and their associated exploitation languages (Section 2). Then, it presents the most relevant PMMSs and discusses their strengths and their limitations according to the metamodeling architecture they support and to their model management capabilities (Section 3). Section 4 devotes a most detailed presentation to the OntoDB/OntoQL PMMS that we have extended to implement our approach. Finally, Section 5 concludes this chapter.

## 2 Model repositories and their exploitation languages

One of the proposed solutions to overcome the problem of scalability faced by classical metamodeling and model management systems running in main memory was to equip these systems with object-relational mapping frameworks (e.g., Hibernate [hib, 2012], EclipseLink [ecl, 2013]) to persist metamodels, models and instances in dedicated databases called *model repositories* or *metadata repositories*. These persistence frameworks play the role of the intermediate between the metamodeling or the model management tool and the model repository.

### 2.1 Model repositories

Several model repositories have been proposed to store large scale models and data. We list below a non exhaustive list of such model repositories.

- CDO [cdo, 2013] is a model repository associated to the Eclipse Modeling Framework (EMF) tool. It supports the storage of EMF metamodels and models and can be used in a distributed environment for persisting metamodels, models and instances. Figure 2.1 illustrates the CDO architecture.

  CDO offers the capability to use different back-end databases to persist metamodels, models and instances such as NoSQL, relational or object databases. Moreover, it provides the possibility of multiuser access to the model repository and ensures the ACID (Atomicity, Consistency, Isolation, Durability) properties for transactions.

- EMFStore [Koegel and Helming, 2010] is a model repository associated to the EMF tool. This model repository has been proposed for collaborative editing of models and for model versioning.

EMFStore possesses an architecture similar to the one of the CDO model repository presented in Figure 2.1. However, EMFStore provides some features that are not available in CDO such as the possibility to work in an off-line mode or to detect the eventual conflicts raised by changing models from different users [emf, 2013b].

- Morsa [Espinazo-Pagán et al., 2011] is a NoSQL-based solution for persisting and accessing very large models. This model repository stores metamodels, models and data and provides the capability to update, delete and query metamodels, models and data using its associated exploitation language. Moreover, Morsa supports incremental storage and update of models, and offers the possibility to partially or totally load models and data. Yet, all the modifications and the model management tasks, that may concern a model stored in Morsa, require loading the model into the modeling tool (evolving in main memory).

### 2.2 Exploitation languages

Model repositories are equipped with declarative query languages (e.g., mSQL [Grant et al., 1993]). These languages are restricted to querying capabilities so that they support neither metamodeling nor model management capabilities. For instance, mSQL [Grant et al., 1993] is a declarative query language for MOF-based model repositories. It provides the capability to define higher-order and model-independent queries. mSQL is restricted to querying capabilities and does not offer any ability to perform model management tasks such as model transformation.

Thus, model repositories query languages remain only high-level query languages since they do not offer the capability to perform operations on models and data. Consequently, persistent model repositories remain simple model warehouses used to store very large models.

In this section we have presented several approaches to equip classical metamodeling and model management tools with a database repository and an exploitation language. In the next section, we study the dual approach which consists in extending a database with metamodeling capabilities. We call these systems *Persistent MetaModeling Systems* (PMMSs) as they provide an extended database environment for metamodeling and model management.

## 3 Persistent metamodeling and model management systems

Several PMMSs have been proposed in the literature. Each PMMS has been set up for specific model management operations. This section lists and analyzes the most relevant ones regarding the metamodeling architecture they support and the model management capabilities they propose.

### 3.1 ConceptBase and GeRoMe

ConceptBase [Jarke et al., 2009b] is an object-oriented and deductive PMMS based on an object-oriented database and the Telos language [Mylopoulos et al., 1990] defined for designing applications. It claims the concept of next generation databases which shall in particular integrate different programming languages (e.g., imperative, declarative) that can be interoperable. This PMMS has been set up for metamodeling and model management.

**Metamodeling architecture**

ConceptBase supports an unlimited metadata hierarchy i.e., the instance level, the model level, the metamodel level, the metametamodel level and so on. This approach is illustrated in Figure 2.2.

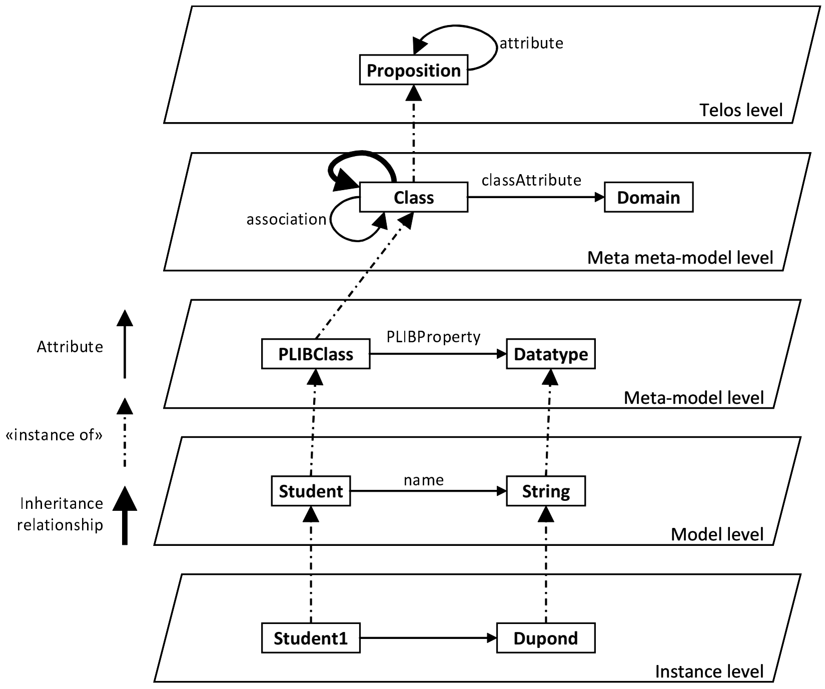


Figure 2.2: metadata hierarchy of ConceptBase

In this example, the *Telos level* defines the concept of `Proposition` which is characterized by a set of attributes. In the *metametamodel level* we define the metametamodel used to set up and describe modeling formalisms. This metametamodel defines the concept of `Class` that is described by a set of attributes (`classAttribute`). A class may have super classes and may also have association relationships with other classes. The *metamodel level* sketches a simple metamodel composed of two classes (`PLIBClass` and `PLIBProperty`) where each PLIB class is characterized by a set of PLIB properties. The *Model level* defines a PLIB model with a PLIB class (`Student`) described by a string PLIB property (`name`). Finally, the *Instance level* defines an instance of the defined model. The textual definition of this example using the Telos language is given below.

```
Concept Class with
    attribute
        isA: Class;
        association: Class;
        classAttribute: Domain
end

Class PLIBClass with
    classAttribute
        PLIBProperty: Datatype
```

```
end

PLIBClass Student with
    PLIBProperty
        name: String
end

Student1 in Student with
    name
        n: 'Dupond'
end
```

*Note.* ConceptBase supports the heterogeneity of models since it is able to manage models expressed with different formalisms.

We have seen through this example that besides the support of an unlimited number of abstraction layers, ConceptBase provides object-oriented constructors to define models at the different levels. Indeed, ConceptBase supports constructors of structural elements (e.g., classes, entities), description elements (e.g., attributes, properties) and the other object-oriented features such as the inheritance and the association relationships. Thus, ConceptBase supports the structural and descriptive semantics of models at the different metadata layers. Now, we will analyze the model management capabilities of this PMMS.

**Model management capabilities**

ConceptBase proposes several mechanisms for model management. Indeed, ConceptBase provides the capability to define deductive rules and views. Moreover, it offers a set of predefined functions such as aggregation functions (e.g., count, avg, max, min), arithmetic functions (e.g., sum), string manipulation functions (e.g., concat), etc.

ConceptBase authorizes the integration of user-defined functions and supports advanced programming features like recursion. The user-defined functions may be defined with membership constraints and may be implemented by external programs. However, the external implementations of the user-defined functions can only be expressed using the Prolog language. Besides, external programs have to be stored in a special and internal file system, and require restarting the server (cold start) in order to support the functions newly introduced or modified [Jeusfeld et al., 2013]. The cold start aspect restricts the availability and the performance of the PMMS.

The following example shows the definition of a function using ConceptBase. This function transforms a PLIB class to an OWL class.

```
PLIBClass2OWLClass in Function isA PLIBClass with
        parameter
            oc: OWLClass
        comment
            c: transforms a PLIB class to an OWL class
        constraint
            ...
end
```

GeRoMe [Kensche et al., 2007] is a PMMS that has been proposed for model management. This PMMS extends ConceptBase with an algebra of atomic model management operators dedicated to model transformation and to import and export models in their native format. In addition, GeRoMe offers the possibility to define derived operators by combining existing ones [Jarke et al., 2009a]. This enables reusing existing operators in multiple model management contexts.

However, GeRoMe did not bring a more flexible programming environment to ConceptBase. Indeed, even if we can introduce user-defined operations (which is allowed by ConceptBase), the implementation of these operations has to be done with the Prolog language and consequently, we cannot get benefit from the power and the coverage of other programming languages nor web services.

## 3.2 Rondo

Rondo [Melnik et al., 2003] is a generic model management system that has been proposed for model mappings. This PMMS is based on a relational database (SQL DBMS) to store models and instances.

**Metamodeling architecture**

Rondo supports a modeling architecture with a hard-coded metamodel layer so that we cannot introduce new modeling formalisms. Rondo supports the structural and descriptive semantics of models using three conceptual constructors:

- *models*: represented by graphs where nodes define models concepts (classes, attributes, etc.) and edges represent relationships between models elements. Each model element is identified with a unique object identifier;
- *morphisms*: define the set of binary mapping relationships between elements of two models which can be useful for data warehousing or data integration;
- *selectors*: represent a set of model elements that can be issued from different models.

**Model management capabilities**

Rondo provides an algebra of *primitive* high-level operators for model management and model mappings such as *Match*, *Delete*, *Extract*, *Domain*, *RestrictDomain* and *Compose*. These operators are hard-coded and thus their implementation is not flexible in the sense that it cannot be done with external programs or web services. Moreover, Rondo supports the definition of derived operators by composing existing ones (e.g., *Range*, *RestrictRange*). Nonetheless, implementing model management tasks such as model transformation with this set of limited operators is not an easy task.

## 3.3 Microsoft repository

Microsoft repository [Bernstein et al., 1999] is a PMMS implemented on the top of a relational database and SQL system. The Microsoft repository has a hard-coded metamodel layer so that it is compatible only with the Microscoft's Component Object Model (COM). Consequently, this PMMS does not support models expressed in other formalisms. Moreover, the Microsoft repository provides the capability to define models and instances, and manage them in the repository. Especially, this PMMS offers the ability to manage reusable model components and to exchange models and data.

## 3.4 Clio

Clio [Hernández et al., 2001] is a PMMS defined for facilitating the tasks of heterogeneous data transformation and integration and model mappings. These tasks are accomplished by mapping a source schema to a target schema using SQL statements. Thus, to accomplish the example which transforms a PLIB class into an OWL class, the usage of SQL is not adapted since it is not an object-oriented language, and it does not provide the capability to create and manipulate (update, delete and query) high-order objects (e.g., classes, metaclasses). Indeed, SQL remains a low-level database exploitation language restricted only to manipulate the database schema and the system catalog.

## 3.5 DB-MAIN

DB-MAIN [Hick and Hainaut, 2003] is a PMMS designed for the management of database evolution. It is based on a fixed hard-coded metamodel and offers a set of built-in high-level operators for modifying the database structure and contents when an evolution is required.

DB-MAIN uses classical mechanisms of databases for model management like triggers, stored procedures and views. It supports some programming features like recursion. Moreover, DB-MAIN provides the capability to store and invoke C++ programs that are pre-compiled and stored in a special file system.

## 3.6 OntoDB/OntoQL

OntoDB/OntoQL [Dehainsala et al., 2007] is a PMMS initially defined for the management of ontologies. It includes the OntoDB model repository which is able to store various metamodels, models and instances, and the OntoQL exploitation language handling metamodeling and model management capabilities.

**Metamodeling architecture**

OntoDB/OntoQL supports a four-layered metamodeling architecture so that it enables the introduction of multiple modeling formalisms. This PMMS provides concepts to define the structural and descriptive semantics of metamodels and models, and supports object-oriented modeling features such as inheritance and association relationships.

**Model management capabilities**

OntoDB/OntoQL uses only the predefined operators and the PgPL/SQL procedural language of its back-end DBMS (PostgreSQL) for managing models and their instances. The usage of the SQL operators and PgPL/SQL is limited to the manipulation of low-level data i.e., simple types. Indeed, PgPL/SQL cannot manipulate complex types (e.g., metaclasses, classes) and does not possess the same coverage as other programming languages (e.g., Java) and web services. Consequently, OntoDB/OntoQL cannot define high-level operators that can manage metamodels and models concepts.

Next section details the OntoDB/OntoQL PMMS we have used for setting up our proposition.

# 4 The OntoDB/OntoQL PMMS

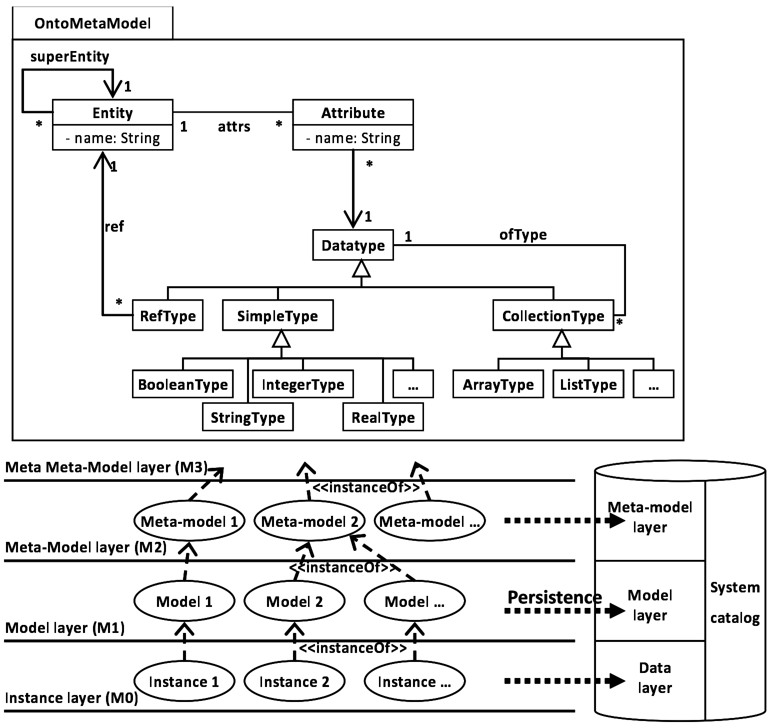


Figure 2.3: The architecture of OntoDB/OntoQL

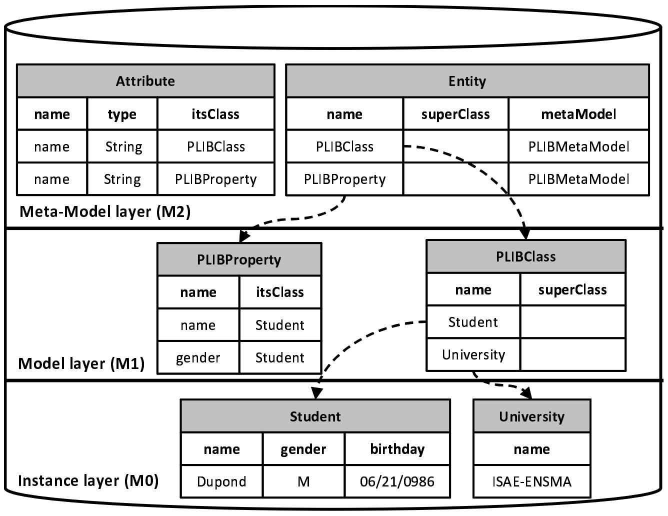
OntoDB/OntoQL is a PMMS that supports a four-layered metamodeling architecture (Figure 2.3). This PMMS includes the OntoDB model repository and the OntoQL metamodeling language.

This section exposes the architecture of the OntoDB model repository. Then, it presents the OntoQL language and its metamodeling and model management capabilities. Finally, this section raises some limitations of the OntoDB/OntoQL PMMS to motivate our proposition.

## 4.1 The OntoDB model repository

### 4.1.1 Architecture

The architecture of the OntoDB repository consists in four parts as shown in Figure 2.3. The *Data layer* and the *System catalog* are the classical parts of traditional databases. The instance part stores instance data while the system catalog stores the descriptions of all structures existing in the database (i.e., tables, views, indexes, etc.). The *Meta-model layer* and the *Model layer* store respectively metamodels and models. OntoDB respects the separation of the different abstraction levels and satisfies the conformity of instances to models, models to metamodels and metamodels to the OntoDB/OntoQL metametamodel.



| Attribute | | |
|---|---|---|
| name | type | itsClass |
| name | String | PLIBClass |
| name | String | PLIBProperty |

| Entity | | |
|---|---|---|
| name | superClass | metaModel |
| PLIBClass | | PLIBMetaModel |
| PLIBProperty | | PLIBMetaModel |

| PLIBProperty | |
|---|---|
| name | itsClass |
| name | Student |
| gender | Student |

| PLIBClass | |
|---|---|
| name | superClass |
| Student | |
| University | |

| Student | | |
|---|---|---|
| name | gender | birthday |
| Dupond | M | 06/21/0986 |

| University |
|---|
| name |
| ISAE-ENSMA |

Figure 2.4: Representing the different metadata layers in OntoDB

### 4.1.2 Representation of the different abstraction layers

OntoDB stores all the concepts of the different abstraction levels in relational tables since this PMMS is based on the PostgreSQL DBMS. Figure 2.4 shows the main tables used to store metamodels, models and data of our example in OntoDB.

The metamodel layer of OntoDB contains two main tables: `Entity` and `Attribute` that store respectively entities and attributes of the different metamodels. For instance, the `PLIBClass` and `PLIBProperty` entities are stored in the `Entity` table whereas attributes describing these two entities are stored in the `Attribute` table.

Each entity of the metamodel layer is associated to a corresponding table, at the model level, where concepts of the model layer are stored. Attributes describing the defined entities are represented by columns in the tables persisting models concepts. For instance, the `PLIBClass` entity is associated to a corresponding table at the model layer to store the different PLIB classes (in our case `Student` and `University` classes). This table has two columns (`name`, `itsClass`) corresponding to the attributes of the `PLIBClass`.

Similarly, each concept of the model layer is associated to a table at the data level to store models instances. For example, the `Student` and `University` PLIB classes are associated to tables that persist instances of these two classes.

## 4.2 The OntoQL exploitation language

Once data of the different abstraction levels are stored in OntoDB, traditional database exploitation languages such as SQL are not suitable for their management. Indeed, these languages require a deep knowledge of the database representation used by the model repository for the different layers of data. In this context, the OntoQL language [Jean et al., 2006b] has been proposed to manipulate metamodels, models and data making abstraction of the database representation used for storing all data.

OntoQL is a declarative and object-oriented language used to create, manipulate (modify, drop) and query metamodels, models and data. In this section, we present the grammar and examples of the OntoQL statements used for defining metamodels, models and data.

### 4.2.1 Metamodel definition

The metamodel part of OntoDB can be enriched to support new metamodels using the OntoQL language. The grammar used to create a new metamodel entity is:

```
<entity definition>    ::= CREATE ENTITY <entity id> [<under clause>] <attribute clause>
<under clause>         ::= UNDER <entity id list>
<attribute clause>     ::= <attribute definition list>
<attribute definition> ::= <attribute id> <datatype>
```

*Example.* The metamodel of our example (Figure 1.4) can be created with the statements of Listing 2.1.

Listing 2.1: Statements for creating metamodels elements

```
CREATE ENTITY #PLIBClass (
              #name STRING,
              #superClass REF (#PLIBClass));

CREATE ENTITY #PLIBProperty (
              #name STRING,
              #domain REF (#PLIBClass));

CREATE ENTITY #PLIBOntology (
              #name STRING,
              #classes REF (#PLIBClass) ARRAY);

...
```

*Meaning.* These statements create the elements of the PLIB metamodel. For instance, the first statement creates the `PLIBClass` and associates two attributes (`name` and `superClass`) to this class.

*Note.* In this statement the # prefix indicates that the definition of an element must be inserted in the metamodel level of OntoDB (an element of the model level does not have a # prefix). Moreover, the `UNDER` keyword denotes the inheritance relationship, and the `REF` keyword the aggregation relationship.

### 4.2.2 Model definition

Once a metamodel is defined, models conforming to that metamodel can be created. The OntoQL grammar used to instanciate an entity of a metamodel is:

```
<class definition>      ::= CREATE <entity id> <class id> [<under clause>] [<descriptor clause>]
                              [<properties clause list>]
<under clause>          ::= UNDER <class id list>
```

```
<descriptor clause>     ::= DESCRIPTOR (<attribute value list>)
<attribute value>       ::= <attribute id> = <value expression>
<properties clause>     ::= <entity id> (<property definition list>)
<properties definition> ::= <prop id> <datatype> [<descriptor clause>]
```

*Example.* The model of our example can be created using the statements defined in Listing 2.2.

Listing 2.2: Statements for creating the model of our example

```
CREATE #PLIBClass University
PROPERTIES (name STRING);

CREATE #PLIBClass Student
PROPERTIES (name STRING,
            gender STRING,
            birthday STRING,
            itsUniversity REF (University));
```

*Meaning.* These two statements create a model conforming to the PLIB metamodel (previously created). The first statement creates the `University` PLIB class characterized by a property (`name`). The second statement creates the `Student` PLIB class described by several properties.

### 4.2.3 Instance definition

Similarly to the previous step, once models have been created with OntoQL and stored in OntoDB, they can be instantiated to create instances. The OntoQL grammar to create an instance of a model element is similar to the SQL one:

```
<insert statement>              ::= INSERT INTO <class id> <insert description and source>
<insert description and source> ::= <from subquery> | <from constructor>
<from subquery>                 ::= [(<property id list>)] <query expression>
<from constructor>              ::= [(<property id list>)] <value clause>
<values clause>                 ::= VALUES (<values expression list>)
```

*Example.* Instances of our example can be created using the statements defined in Listing 2.3.

Listing 2.3: Statements for creating instances

```
INSERT INTO University
VALUES ('ISAE-ENSMA');

INSERT INTO Student
VALUES ('Dupond', 'M', '06/21/1986', 123);
```

*Meaning.* The first statement creates an instance of the `University` class while the second one defines an instance of the `Student` class.

This presentation of OntoQL shows its capability to define metamodels, models and instances with modeling features such as the inheritance relationship. OntoQL supports as well the other basics tasks like accessing, modifying and deleting metamodels, models and data.

## 4.3 Limitations of OntoDB/OntoQL

The OntoDB/OntoQL PMMS provides the capability to use only the user-defined stored procedures (PL/pgSQL procedures) and the predefined operators of its back-end DBMS (PostgreSQL) to manage models and data. For instance, the OntoQL statement of Listing 2.4 uses the `CONCAT` operator, predefined in SQL and provided with the DBMS, to concatenate the `firstname` and the `lastname` properties of the `Student` class.

Listing 2.4: Statements for creating the model of our example

```
SELECT CONCAT(firstname, lastname)
FROM Student;
```

Currently, the OntoDB/OntQL does not provide any mechanism for high-level model management. Indeed, the stored procedures and the predefined operators do not support the manipulation of high-level concepts (e.g., classes, metaclasses) since they support only simple types data (e.g., string, integer). Let us consider, for example, the definition of an OWL class as the result of the transformation of a PLIB class. Here appears the need of computing a new concept from an existing one. Such a construction is not available in OntoDB/OntoQL as it does not provide the capability to introduce dynamically model management operations that manipulate such high-level concepts.

# 5 Conclusion

With the development of large models, the scalability of MDE is becoming a crucial challenge. In this chapter we have overviewed the efforts made to address this challenge using database technologies. Currently most of the efforts have been made to improve metamodeling systems evolving in main memory by associating them to database repositories. The data necessary for a model management operation is loaded into the metamodeling system which performs the desired operation. Thus, metamodeling systems must usually develop their own optimizations to perform model management operations on large models. The reverse approach followed by PMMSs consists in extending databases with model management capabilities. But currently, the functionalities offered by existing PMMSs are limited. Our idea is that PMMSs shall provide advanced capabilities for metamodeling and model management such as those provided by classical metamodeling and model management tools evolving in main memory. Particularly, we focus on the capability of PMMSs to define model management operations that can be implemented using different mechanisms like external programs or web services. This capability is neither covered by existing PMMSs nor partially fulfilled. Thus, in the next part of this thesis, we address the extension of PMMSs with the capability to handle model management operations.

# Part II

# Contributions

Chapter 3

# Requirements for complete persistent metamodeling systems

**Contents**

**Abstract.** In the first part of this thesis, we have introduced some basic notions around metamodeling and model management. Then, we have presented existing PMMSs and discussed their advantages and limitations. Based on this study, we present in this chapter the requirements we have defined for complete PMMSs. These requirements combine benefits of classical metamodeling and model management systems together with those of database environments.

# 1 Introduction

As claimed throughout this thesis, our objective is to have a metamodeling and model management system (1) satisfying metamodeling and model management features, and (2) handling the capability to store and to manipulate large-scale and heterogeneous models and data. We observed that a database environment enabling metamodeling and model management characteristics could fit with our needs in terms of metamodeling and model management. We called this type of systems *Persistent MetaModeling Systems* (PMMSs).

In this sense, a PMMS shall satisfy metamodeling features since it shall enable expressing the structural semantics of metamodels and models (e.g., by providing factories of classes, metaclasses, entities, etc.). As well, a PMMS shall provide elements to describe these structural concepts. This can be done using, for example, attributes, metaattributes, properties, etc. These elements are supported by classical metamodeling systems evolving in main memory such as Eclipse Modeling Framework (EMF) [emf, 2013a].

Another aspect that a PMMS shall satisfy is the capability to manage models and data in the same environment. Indeed, models and data can be subjects to different manipulations such as transformation (e.g., UML class diagram to relational database scheme), code generation (e.g., generating the Java or C++ code of an UML class diagram), storing, annotating, archiving, retrieving, etc. Basically, several dedicated tools, languages and platforms have been set up in order to accomplish such model management tasks (e.g., ATL [Jouault and Kurtev, 2005] for model transformation, Acceleo [acc, 2013] for code generation).

At this level, all these aspects cited so far, which are metamodeling and model management capabilities, are supported by existing tools evolving in main memory. However, knowing that models and data in real contexts, especially in industrial ones, may be composed of thousands of classes and properties, and may describe millions of data instances, the modeling and model management tools using main memory may not handle these over-sized models and data. Moreover, exchange files may also be not efficient enough for exchanging large scale models and data, and consequently we need a common repository for sharing models and data. These features are available in database systems which can offer at the same time the scalability and a common platform to share data and models.

Thus, our objective is to gather strengths of classical metamodeling and model management tools together and those of database environments in order to provide a complete PMMS supporting all metamodeling and model management features. Moreover, the PMMS shall ensure the scalability, support the heterogeneity of models and data and serve as a common repository for sharing models and data. Furthermore, a PMMS shall satisfy additional technical requirements related to the flexibility and to the performance such the hot start criterion ensuring the permanent availability of the system. These technical aspects are important in the sense that they will give PMMSs more flexibility and a further expressive power.

In this chapter we define the requirements for a complete PMMS (Section 2). Each requirement is presented and justified. Then, we analyze existing PMMSs presented in Chapter 2 according to the defined requirements (Section 3). In Section 4, we introduce the objective of our work in this thesis in order to fulfill the defined requirements. Finally, Section 5 concludes this chapter.

## 2 Requirements for complete PMMS

### 2.1 Database persistence

**Requirement 1**

*A PMMS shall offer a database environment to store and to manipulate different metamodels, models and instances while respecting the separation of the different abstraction layers and preserving the conformity of models to metamodels and instances to models. Moreover, a PMMS shall be equipped with an exploitation language enabling the definition and the manipulation, of metamodels, models and instances.*

**Justification**

Over-sized metamodels, models and instances should be stored and managed in a database environment. Designed carefully, the database guarantees the absence of redundancy, the integrity, the confidentiality and the continuity of data. In our work we focus on two aspects.

- *Scalability*: database systems are the best infrastructure that can support the efficient storage of large-scale models and instances. This approach has been followed for specific models such as ontologies. Indeed, multiple dedicated databases called *semantic databases* or *ontology-based databases* (e.g., OntoDB [Dehainsala et al., 2007]) have been set up to store and manipulate large ontologies and instances efficiently and in a scalable way. Thus, by using a database, we can partially load models and/or instances in main memory to process them using the query optimization engine of the DBMS.

- *Sharing models and data*: database environments offer a common repository for sharing models and data. Moreover, DBMSs offer mechanisms for ensuring all security properties in order to avoid potential conflicts of data modification. Nowadays, existing DBMSs are enough mature to secure the access and the manipulation of data.

### 2.2 Extensible metamodel layer

**Requirement 2**

*A PMMS shall offer an extensible metamodel layer so that multiple modeling formalisms can be supported. Moreover, a PMMS shall conform to a metamodeling architecture standard in order to guarantee aspects such as the interoperability and the portability of models and data. Nowadays, the Meta-Object Facility (MOF) [mof, 2011] architecture is a standard adopted in most of the current metamodeling and model management systems. Thus, we suggest that a metamodeling system should support the four-layered MOF architecture which offers an extensible metamodel layer.*

**Justification**

Software engineering uses a lot of different models (e.g., entity-relationship, functional or state transition models). Thus, a PMMS shall support the definition of an unlimited number of metamodels.

## 2.3 Support of structural and descriptive semantics

**Requirement 3**

*The PMMS exploitation language shall support the definition of structural and descriptive semantics of metamodels and models elements. Indeed, the PMMS exploitation language shall provide constructors of structural elements of models and metamodels. In particular, it shall support the definition of classes, metaclasses, entities, metaentities, etc. of metamodels and models. Furthermore, it shall enable the definition of descriptive elements to characterize structural concepts. These descriptive elements could be attributes, metaattributes, properties, metaproperties, etc. Moreover, the PMMS exploitation language shall provide primitives to express the other object-oriented modeling features such as inheritance and association relationships.*

**Justification**

Following the MOF specification, most models and metamodels can be expressed with object-oriented constructors.

*Example.* In our example of Chapter 1 (Figure 1.4), the PLIB and OWL metamodels are composed of a set of classes that are described by attributes. A simple inheritance relationship is permitted in the PLIB language while multiple inheritance is authorized in the OWL language.

## 2.4 Support of behavioral semantics

**Requirement 4**

*The PMMS exploitation language shall provide the capability to introduce operations (functions, procedures) on the fly. Particularly, these operations shall be able to manipulate models and instances. Besides, the PMMS shall offer the possibility to invoke the defined operations using the PMMS exploitation language.*

**Justification**

Operations on models and instances are important to accomplish advanced model and data management tasks such as model transformation or code generation, checking constraints on models elements, storing, archiving, retrieving, etc.

*Example.* In our example of Chapter 1 (Figure 1.4), an operation could be defined to transform PLIB models to OWL ones. Other operations could also be set up, for instance, to export the OWL models into the XML format, or to compute derived classes or properties. For example, the `age` property of the `Student` class could be computed using the `computeAge` operation. Another operation (`unionOf`) could be defined in order to compute the union of OWL classes.

### 2.5 Flexible programming environment

**Requirement 5**

*A PMMS shall provide an heterogeneous programming environment to implement operations. Particularly, a PMMS shall be able to use existing external programs written in any language (e.g., Java, C++) or web services in order to profit from their coverage and completeness. This aspect brings large flexibility to metamodeling systems and a strong power of expressiveness.*

**Justification**

As it is better to reuse existing pieces of software instead of rewriting them, a PMMS should be able to integrate existing implementations of operations whatever is the programming language used.

*Example.* It is easy to find an existing code that exports an OWL model in XML. Thus, a PMMS should allow users to reuse (to envelop) this piece of software to implement an operation.

### 2.6 Hot-plug of implementations

**Requirement 6**

*A PMMS shall support an immediate usage of the external or remote implementations of operations without restarting the system (warm start). Moreover, a PMMS should not constrain users to store implementations in a specific file system.*

**Justification**

Restarting the PMMS should be avoided for high availability applications. Thus, the definition of an implementation for an operation should not require restarting the PMMS (warm start).

## 3 Synthesis and discussion

Table 3.1: Synthesis of the state of the art

| | Req. 1 | Req. 2 | Req. 3 | Req. 4 | Req. 5 | Req. 6 |
|---|---|---|---|---|---|---|
| **ConceptBase** | Yes | restricted | Yes | restricted | restricted | No |
| **GeRoMe** | Yes | restricted | Yes | restricted | restricted | No |
| **Rondo** | Yes | No | Yes | hard-coded | No | No |
| **Clio** | Yes | No | No | restricted | No | No |
| **DB-MAIN** | Yes | No | Yes | restricted | No | No |
| **OntoDB/OntoQL** | Yes | Yes | Yes | restricted | No | No |

As the previous overview of the state of the art (presented in Chapter 2) shows, each one of the existing PMMSs present some strengths and some limitations regarding the requirements we have defined in this chapter. The identified limitations are presented in Table 3.1. A result of our study is that existing

PMMSs focus mainly on the support of structural and descriptive semantics of metamodels and models, while they pay less attention to behavioral semantics which is essential for model management. Hence, PMMSs should offer resources to be extended in order to support behavioral semantics offering advanced features.

## 4 Objectives

Our objective is to define a PMMS satisfying all the requirements defined in this chapter. Although, some existing PMMSs support some of the defined requirements, we focus in our thesis on the requirements that are not sufficiently addressed.

### 4.1 Handling behavioral semantics

#### Definition of model management operation

Practically, we would like to extend PMMSs to be able to define model management operations with statements that might look like the one of Listing 3.1.

Listing 3.1: Statement for creating a model management operation

```
CREATE OPERATION #PLIBClass2OWLClass
INPUT (REF (#PLIBClass))
OUTPUT (REF (#OWLClass));
```

*Meaning.* This statement is supposed to create a model management operation that transforms a PLIB class to an OWL one. This operation takes as input a PLIB class and returns an OWL class.

#### Definition of data management operation

We also would like to extend PMMSs with the support of data management operations with statements that might look like the one of Listing 3.2.

Listing 3.2: Statement for creating a data management operation

```
CREATE #OPERATION computeAge
INPUT (REF (Person))
OUTPUT (INTEGER);
```

*Meaning.* This statement is supposed to create a data management operation that computes the age of the `Person` class. This operation takes as input a the `Person` class and returns an integer value.

#### Exploitation of the defined operations

Moreover, we would like to be able to exploit the defined operations within the PMMS exploitation language i.e., to invoke the defined operations using the PMMS exploitation language.

*Example.* Exploitation examples of the operations defined above are given in Listing 3.3.

Listing 3.3: Example of exploiting operations

```
CREATE #OWLClass O_Person
AS (SELECT #PLIBClass2OWLClass(P)
    FROM #PLIBClass AS P
    WHERE P.#name = 'Person');

SELECT computeAge(P)
FROM Person;
```

*Meaning.* The first statement is supposed to create an OWL class (`O_Person`) by transforming the `Person` PLIB class using the `PLIBClass2OWLClass` operation. The second statement is supposed to compute the age of all the instances of the `Person` class.

Supporting these behaviors by a PMMS should satisfy only requirement 4 which is related to the support of operations for model and data management.

## 4.2 Flexibility of implementation

In order to satisfy requirement 5, an operation shall be implemented by a program internal to the database system (e.g., a PL/SQL stored procedure) or external to the PMMS. For instance, by using external programs written in a given programming language (e.g., Java, C++) or web services. An implementation of an operation should be defined with a statement looking like the one of Listing 3.4.

Listing 3.4: Example of defining an implementation of an operation

```
CREATE IMPLEMENTATION #PLIBClass2OWLClassJavaImp
DESCRIPTORS (type = 'java',
             location = 'http://.../programs.jar',
             class = 'fr.ensma.lias.myClass',
             method = 'PLIBClass2OWLClass')
IMPLEMENTS #PLIBClass2OWLClass;
```

*Meaning.* This statement associates a Java implementation to the `PLIBClass2OWLClass` operation (previously defined). It gives the necessary metadata (of the Java program) for the execution of that program. In particular, it provides the location of the Java archive (JAR) file where the external program is defined, the class name where the method is defined and the method name.

This extension is close to the notion of *Interface* of the Java programming language. Indeed, the definition of an operation with a PMMS is similar to the definition of the signature of an interface. Likewise, the definition of an associated implementation to a PMMS operation is equivalent to the definition of a method of a Java class implementing a Java interface. Besides, our proposition brings to PMMSs the capability to define heterogeneous implementations (external programs and web services).

## 4.3 Warm start

Finally, to support requirement 6, the introduction of a new operation with a PMMS shall not require restarting the system (cold start) as it will affect its performance. Indeed, web services will be invoked from the PMMS without requiring to restart it. As well, external programs will be loaded automatically to the PMMS each time they are invoked without restarting the PMMS (warm start). Next chapters show how the objectives defined in this section are reached.

## 5 Conclusion

In this chapter, we have defined requirements for complete PMMSs. These requirements gather at the same time metamodeling and model managements features and benefits of database systems for an efficient storage and exploitation of large scale and heterogeneous metamodels, models and instances. The analysis of the studied PMMSs according to the defined requirements concludes to that these PMMSs do not meet all these requirements. Thus, PMMSs have to be extended in order to fulfill the missing requirements, especially the behavioral semantics. This proposal has been validated in [Bazhar, 2012, Bazhar et al., 2013a].

Next chapter presents the formal extension of PMMSs to support behavioral semantics. This proposition includes the extension of the PMMS metametamodel and metamodel layers, the logical extension of the PMMS model repository, and the extension of the algebra of the PMMS exploitation language.

Chapter 4

# BeMoRe : modeling and formal representation

**Contents**

**Abstract.** In the previous chapter, we have defined the requirements for complete PMMSs. In this chapter, we present the formal extension of PMMSs in order to fulfill the defined requirements. Indeed, this chapter addresses (1) the extension of the PMMS metametamodel and metamodel layers to support model and data management operations, (2) the logical extension of the PMMS model repository to store operations signatures and implementations descriptions, and (3) the extension of the algebra of the PMMS exploitation language to support the definition and the exploitation of operations.

## 1 Introduction

As claimed in Chapter 3, a PMMS shall support a set of requirements that cover different aspects. These requirements concern mainly (i) the support of an extensible metamodeling architecture to enable different modeling formalisms, (ii) the support of structural and descriptive semantics to express the static part of metamodels and models, and finally (iii) the support of heterogeneous and flexible behavioral semantics to support operations that can be useful to perform model management tasks suchs as model transformations or code generation. An operation may be implemented using different mechanisms like internal stored procedures (e.g., PL/SQL procedures), external programs (e.g., Java programs) or web services.

Through an overview of the state of the art, we have seen that existing PMMSs (presented in Chapter 2) do not fulfill all the requirements defined in Chapter 3 since they mainly focus on the support of structural and descriptive semantics of metamodels and models. Moreover, existing PMMSs do not handle sufficiently behavioral semantics (in terms of our requirements) neither in their data models nor in their exploitation languages.

Our objective is to support all the requirements defined in Chapter 3 in PMMSs. This includes the integration of the missing requirements in the data model of the PMMS and its associated exploitation language. In our case we have chosen to base our approach on the OntoDB/OntoQL PMMS that was originally defined in our laboratory to manage ontologies and their instances. This PMMS, as it is presented in Chapter 2, does not fulfill the set of requirements we have defined and in particular requirements 4, 5 and 6. Thus, we have extended the data model of OntoDB and the algebra of its associated exploitation language (OntoQL) so that all the defined requirements can be supported.

The support of the defined requirements by PMMSs goes through three parts. The first one, presented in Section 2, concerns the extension of the PMMS conceptual model with new elements to support particularly model management operations as most of PMMSs have a data model supporting only the structural and descriptive semantics of metamodels and models. The second part, described in Section 3, addresses the logical extension of the PMMS model repository in order to support the storage of the signatures of operations and the descriptions of their associated implementations. Then, we describe the formal extended data model of PMMSs in set theory (Section 4). The third part (Section 5) relates to the extension of the algebra of the PMMS exploitation language with operators supporting the definition (create and delete) and the exploitation of model management operations. Although our proposition has been implemented on the OntoDB/OntoQL PMMS, this proposition is generic in the sense that it can be supported by any PMMS. Indeed we present in this chapter the different steps to extend a PMMS with behavioral semantics.

## 2 Conceptual extension

Figure 4.1 gives an overview of the extended PMMS data model we propose. Our model defines the concept of metamodel that includes a set of classes which are described by attributes, and single class inheritance relationships are allowed. This part of our data model is usually available in all PMMSs which, as we have seen in Chapter 2, focus mainly on the structural and descriptive semantics of metamodels

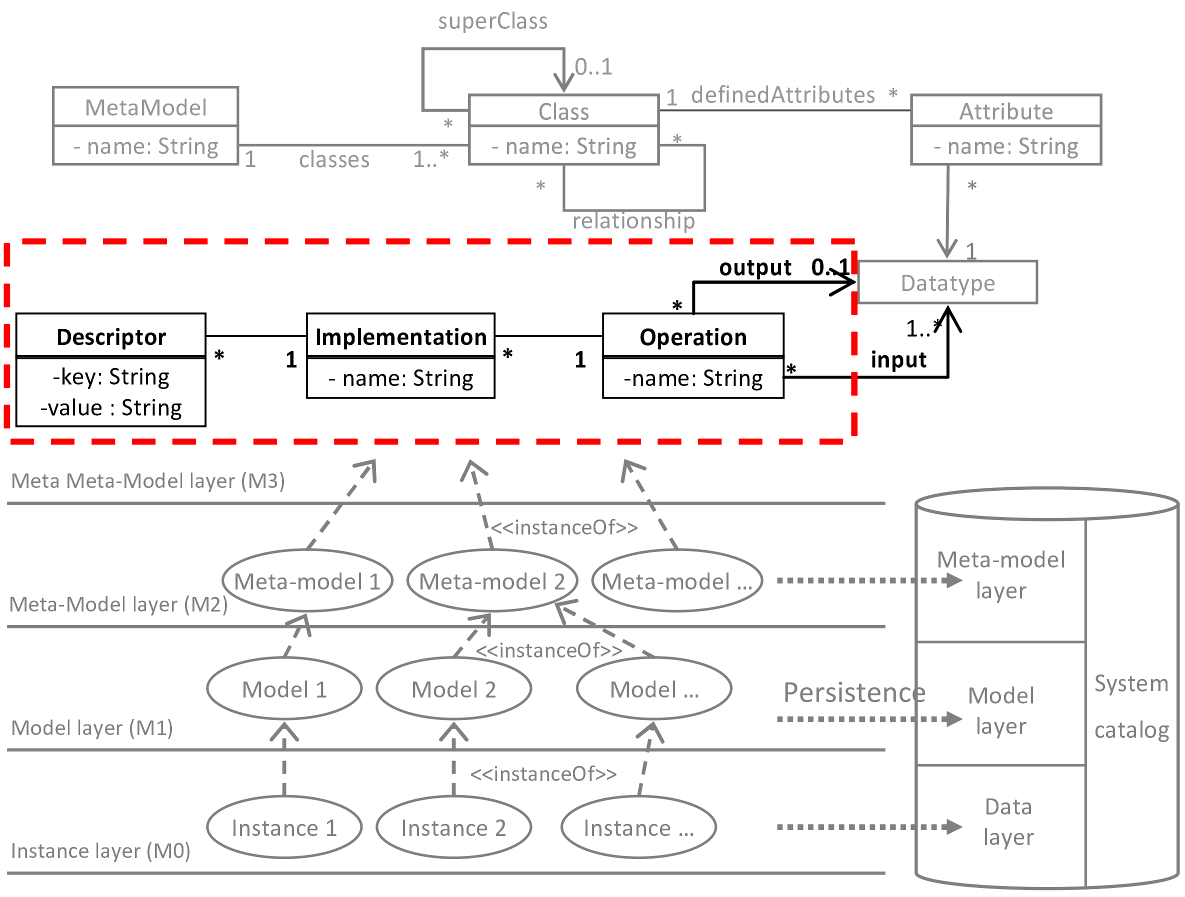


Figure 4.1: The proposed PMMS data model

elements. The dashed area of the metametamodel layer in Figure 4.1 gives an overview of our proposed extension for the definition of behavioral semantics. Our model supports the definition of operations with a list of input and an output data types. Furthermore, an operation can be associated to multiple implementations. Each implementation is itself described by a set of descriptors (pairs of *key*, *value*). With these generic set of descriptors, different programming environments can be easily integrated in our approach.

The extension achieved at the metametamodel layer enables the support of model management operations dedicated to manipulate only models elements. This kind of operations can be used to process operations only on models such as model transformations. Thus, to be able to manipulate instances elements, we have reproduced the same extension (made at the metametamodel layer) at the metamodel level in order to support operations for data management. Indeed, data management operations can be used to achieve tasks such as data transformation or data integration.

The metamodel level defines metamodels as instances of the metametamodel. Similarly, the model level defines models as instances of metamodels. Each layer of the architecture presented in Figure 4.1 is represented in the PMMS model repository. The metametamodel layer is hard-coded in term of tables. Indeed, each concept of the metametamodel is represented by a table to store metamodels elements where each attribute is represented by a column. Similarly, each concept of the metamodel layer is associated to a table to store concepts of the model level. Finally, each concept of the model level is associated to a table to save data of the instance layer.

Once the conceptual elements, that are necessary for metamodeling and model management, are defined, we need to extend the logical metametamodel and the logical metamodel of the PMMS model repository with structures (tables) to store the signatures of the defined operations and the descriptions of their associated implementations. Next section gives the details of the extension of the PMMS model repository.

## 3 Logical extension

The logical extension of the PMMS model repository consists in adding new structures (tables) in order to store the signatures of operations and their associated implementations. Two logical extension are distinguished:

- the extension of the logical metametamodel to handle the storage of model management operations operating at the model level. These operations can be, for instance, model transformation operations, code generation operations, etc.;

- the extension of the logical metamodel level to handle the storage of data management operations evolving at the data level. These operations can be used for example to migrate instances, compute derive properties, transform data, etc.

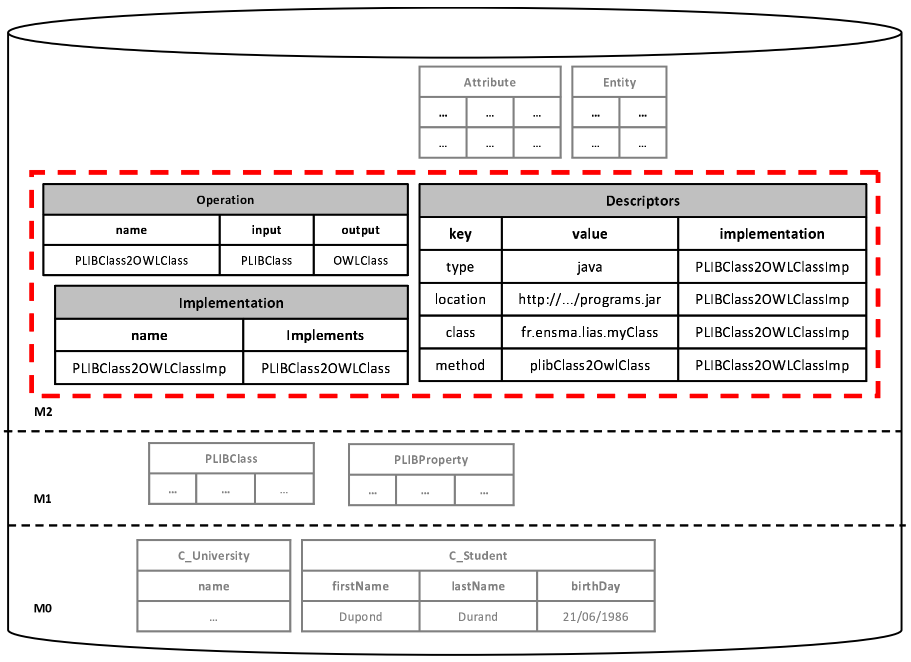


Figure 4.2: Extension of the logical metametamodel of the OntoDB architecture

## 3.1 Extension of the logical metametamodel

Figure 4.2 represents the extension of the logical metametamodel of the OntoDB architecture in terms of relational tables (dashed part). This extension adds 3 main tables to the metamodel layer of OntoDB (as showed in the dashed area): `Operation`, `Implementation` and `Descriptor` which store respectively signatures of operations, implementations and implementations descriptions.

### 3.1.1 Operation table

The `Operation` table stores the signatures of the defined operations. Particularly, it stores names of the defined operations and their input and output data types. This table contains 3 attributes (columns):

- the `name` attribute represents the name of an operation. This attribute is used as the identifier of an operation;
- the `input` attribute represents the input parameters data types of an operation;
- the `output` attribute represents the return data type of an operation.

In the example of Figure 4.2, the `Operation` table stores the signature of the `PLIBClass2OWLClass` operation i.e., its name, its input and output data types (respectively `PLIBClass` and `OWLClass`).

### 3.1.2 Implementation table

The `Implementation` table indicates the implementation relationship between an operation and an implementation. This table contains 2 attributes:

- the `name` attribute represents the name of an implementation. This attribute is used as the identifier of an implementation;
- the `implements` attribute specifies the operation to which the current implementation is associated.

In the example of Figure 4.2, the `Implementation` table indicates that the `PLIBClass2OWLClass-Imp` implementation implements the `PLIBClass2OWLClass` operation.

### 3.1.3 Descriptor table

The `Descriptor` table stores metadata of programs implementing operations. This table contains 3 attributes:

- the `implementation` attribute represents the name of an implementation;
- the `key` attribute specifies an attribute of an implementation type;
- the `value` attribute represents the value of the key attribute.

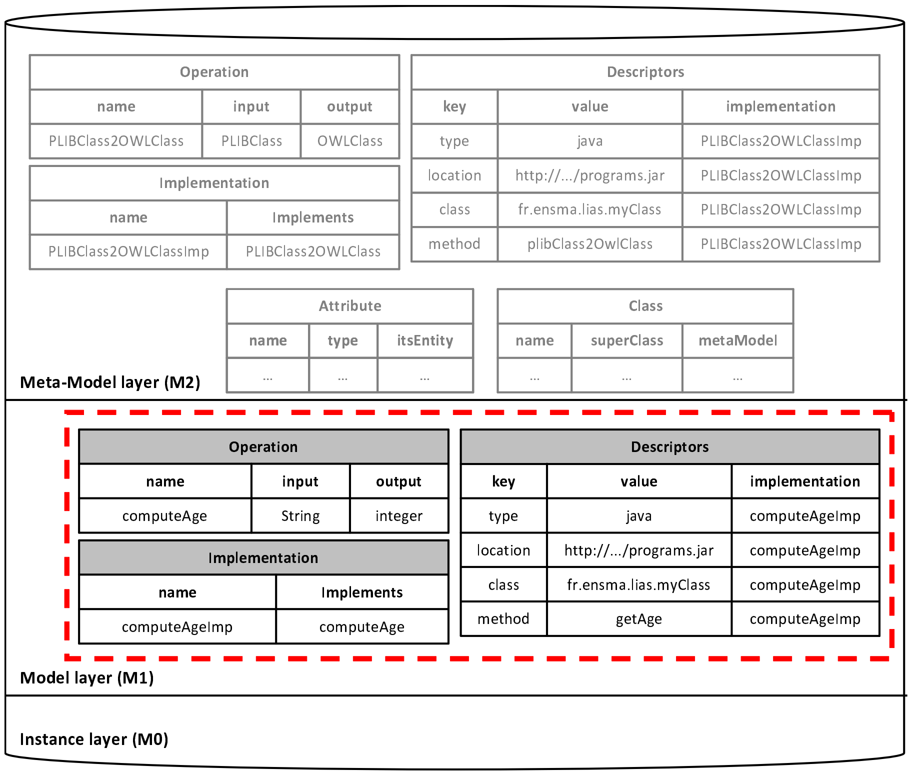
In the example of Figure 4.2, the `Descriptor` table stores descriptors of the `PLIBClass2OWLClass-Imp` implementation. Here `PLIBClass2OWLClassImp` is an external Java program described by 4 attributes: `type` defines the implementation type which is in this example a Java program, `location` specifies the location of the file containing the Java program, `class` characterizes the class name containing the method to execute and finally `method` which represents the name of the Java method that will be executed.

*Note.* As a future work, we can envision to associate a program (instance of a metaprogram or a BNF) that could be interpreted. We will address this point in the perspectives of this work.

## 3.2 Extension of the logical metamodel

Figure 4.3 represents the extension of the logical metamodel of the OntoDB architecture expressed in terms of relational tables (the dashed part). Similarly to the extension of the logical metametamodel, this extension adds the corresponding tables to the model layer of OntoDB (as shown in the dashed area).

Notice that the metamodel and the model parts of OntoDB store different types of operations. Indeed, the metamodel part contains model management operations handling models elements and operating at the model level, while data management operations stored at the model level act at the data layer and



| Operation | | |
|---|---|---|
| name | input | output |
| PLIBClass2OWLClass | PLIBClass | OWLClass |

| Implementation | |
|---|---|
| name | Implements |
| PLIBClass2OWLClassImp | PLIBClass2OWLClass |

| Descriptors | | |
|---|---|---|
| key | value | implementation |
| type | java | PLIBClass2OWLClassImp |
| location | http://.../programs.jar | PLIBClass2OWLClassImp |
| class | fr.ensma.lias.myClass | PLIBClass2OWLClassImp |
| method | plibClass2OwlClass | PLIBClass2OWLClassImp |

| Attribute | | |
|---|---|---|
| name | type | itsEntity |
| ... | ... | ... |

| Class | | |
|---|---|---|
| name | superClass | metaModel |
| ... | ... | ... |

| Operation | | |
|---|---|---|
| name | input | output |
| computeAge | String | integer |

| Implementation | |
|---|---|
| name | Implements |
| computeAgeImp | computeAge |

| Descriptors | | |
|---|---|---|
| key | value | implementation |
| type | java | computeAgeImp |
| location | http://.../programs.jar | computeAgeImp |
| class | fr.ensma.lias.myClass | computeAgeImp |
| method | getAge | computeAgeImp |

Figure 4.3: Extension of the metamodel level of the OntoDB architecture

relate to instances. For example, `PLIBClass2OWLClass` is a model management operation whereas `computeAge` is a data management operation.

Next section presents the formalization of the PMMS data model.

## 4 Formalization of the conceptual extension

The obtained extension of PMMSs can be formally defined by the $< MM, CL, ATT, DT, OP, IMP, Desc, M, Inst >$ structure where $MM$, $CL$, $ATT$, $DT$, $OP$, $IMP$, $Desc$, $M$, $Inst$ are respectively sets of metamodels, classes, attributes, data types, operations, implementations, implementation descriptors, models and models instances. Next subsections give the definition of each one of these sets as well as rules concerning these sets described in set theory.

*Note.* We used the EXPRESS language [Pierra et al., 1995] to represent the proposed model. Unlike some languages, such as the B language, where the nature of any function must be specified, the EXPRESS language considers all functions total, and does not support other types of functions (e.g., partial, surjective). Thus, to complete the semantics of our model, the EXPRESS language provides the ability to define constraints to enrich the model.

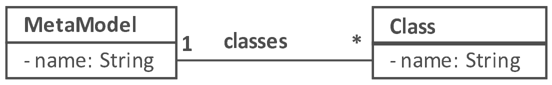


Figure 4.4: A metamodel is composed of a set of classes

## 4.1 Metamodels

A metamodel corresponds to a modeling formalism. It offers constructors of structural and descriptive concepts and defines rules and constraints that must be respected when designing models. Indeed, metamodels must have a unique name and a set of classes. These rules are formally defined in Table 4.1.

Table 4.1: Main rules of metamodels

| |
|---|
| **A metamodel is characterized by a unique name:**<br>$mmName : MM \rightarrow String$<br>Where $\rightarrow$ represents a total function<br>$\forall (mm_i, mm_j) \in MM, i \neq j \Rightarrow mmName(mm_i) \neq mmName(mm_j)$ |
| **A metamodel has a set of classes (see Figure 4.4):**<br>$classes : MM \rightarrow P(CL)$ |

### 4.1.1 Classes

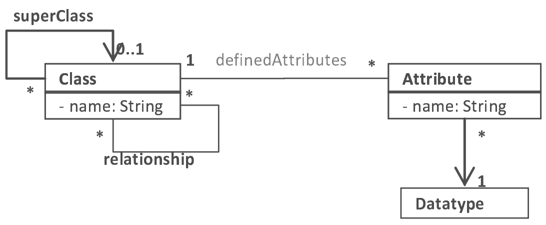


Figure 4.5: The `Class` and `Attribute` concepts

A class (see Figure 4.5) represents concepts having the same characteristics. A class is described by a set of attributes and is subject to several rules such as the uniqueness of the name as the class name represents the identifier of the class in a metamodel. Moreover, a class may have at most one superclass since only a simple inheritance relationship between classes is authorized. This relationship implies that the subclass inherits attributes of the superclass. Furthermore, the subclass may define its own attributes. These rules are formally defined in Table 4.2.

### 4.1.2 Attributes

An attribute is a description element that defines a characteristic of a class. An attribute belongs to a class and is characterized by a data type. Moreover, the attribute concept is as well subject to some rules such as the uniqueness of the name in a class. These rules are described in Table 4.3.

Table 4.2: Main rules of classes

| |
|---|
| **A class belongs to a unique metamodel.** <br> $\forall(mm_i, mm_j) \in MM, i \neq j \Rightarrow classes(mm_i) \cap classes(mm_j) = \emptyset$ |
| **A class is characterized by a unique name in a metamodel.** <br> $clName : CL \rightarrow String$ <br> $\forall mm \in MM, \forall(cl_i, cl_j) \in CL, i \neq j, (cl_i, cl_j) \in classes(mm) \Rightarrow$ <br> $clName(cl_i) \neq clName(cl_j)$ |
| **A class may have at most one superclass (single inheritance relationship).** <br> $superClass : CL \nrightarrow CL$ <br> Where $\nrightarrow$ is a partial function |
| **A class may inherit attributes from its super class.** <br> $inheritedAttributes : CL \rightarrow P(ATT)$ <br> $\forall cl \in CL$, if $superClass(cl) \in CL$ then: $inheritedAttributes(cl) = attributes(superClass(cl))$ <br> else: $inheritedAttributes(cl) = \emptyset$ <br> The *attributes* function is defined below |
| **A class may be described by additional attributes.** <br> $definedAttributes : CL \rightarrow P(ATT)$ <br> $\forall cl \in CL \Rightarrow definedAttributes(cl) \cap inheritedAttributes(cl) = \emptyset$ <br> $\forall(cl_i, cl_j) \in CL, i \neq j \Rightarrow definedAttributes(cl_i) \cap definedAttributes(cl_j) = \emptyset$ |
| **The set of attributes of a class.** <br> $attributes : CL \rightarrow P(ATT)$ <br> $\forall cl \in CL \Rightarrow attributes(cl) = inheritedAttributes(cl) \cup definedAttributes(cl)$ |

Table 4.3: Main rules of attributes

| |
|---|
| **An attribute is characterized by a unique name in a class.** <br> $attName : ATT \rightarrow String$ <br> $\forall cl \in CL, \forall(att_i, att_j) \in ATT, i \neq j, (att_i, att_j) \in attributes(cl) \Rightarrow$ <br> $attName(att_i) \neq attName(att_j)$ |
| **An attribute has a data type.** <br> $typeOf : ATT \rightarrow DT$ |

## 4.2 Datatypes

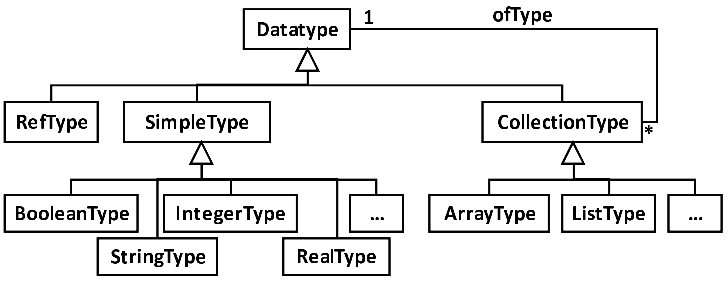


Figure 4.6: The PMMS data types system

Table 4.4: Main rules of data types

| |
|---|
| $SimpleType = BooleanType \cup IntegerType \cup StringType \cup RealType$<br>$CollectionType = ArrayType \cup ListType$<br>$DT = RefType \cup SimpleType \cup CollectionType$<br>$RefType \cap SimpleType = \emptyset$<br>$RefType \cap CollectionType = \emptyset$<br>$CollectionType \cap SimpleType = \emptyset$ |
| **A `RefType` references one unique class.**<br>$ref : RefType \rightarrow CL$ |
| **A `CollectionType` is characterized by a unique datatype.**<br>$typeOfCollection : CollectionType \rightarrow DT$ |

Figure 4.6 represents the data types system that shall be supported by a PMMS. For the purpose of this thesis, we consider the data types system of the OntoDB/OntoQL PMMS on which we have based our approach. This data types system includes simple types (i.e., string, integer, boolean and real), the complex type that references a class of a metamodel and the aggregate type which may be an array, a list, a bag, etc. of a data type. We assume that the considered data types system are gathered in the set *DT* defined in Table 4.4.

## 4.3 Model management operations

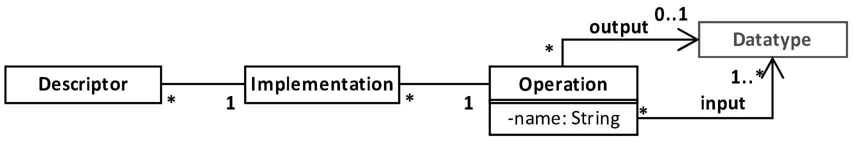


Figure 4.7: Elements extending the PMMS data model to handle behavioral semantics

Figure 4.7 represents the extension of the basic PMMS data model. This extension consists in three main classes detailed in the following subsections.

### 4.3.1 Operations

An operation is a model management function or procedure dedicated to accomplish advanced computations like model transformation, code generation, derivation, etc. An operation is characterized by a unique name as this attribute represents the identifier of the operation. Moreover, an operation is characterized by parameter types representing the input of the operation, and the return type corresponding to the output of the operation. These rules are formally defined in Table 4.5.

Table 4.5: Main rules of operations

| |
|---|
| **An operation is characterized by a unique name.**<br>$opName : OP \to String$<br>$\forall (op_i, op_j) \in OP, i \neq j \Rightarrow opName(op_i) \neq opName(op_j)$ |
| **An operation has a list of input parameters.**<br>$input : OP \text{ x } \mathbb{N}^+ \to DT$ |
| **An operation returns a result or void.**<br>$output : OP \to DT \cup void$ |

### 4.3.2 Implementations

Table 4.6: Main rules of implementations

| |
|---|
| **An operation may have several implementations.**<br>$implementations : OP \to P(IMP)$<br>$\forall (op_i, op_j) \in OP, i \neq j \Rightarrow implementations(op_i) \cap implementations(op_j) = \emptyset$ |

Each operation can be implemented by one or many implementations. An implementation must satisfy the rules of Table 4.6.

### 4.3.3 Descriptors

Each implementation is described by a set of descriptors that indicate the necessary metadata for the execution of the program. An implementation is concerned by the rules of Table 4.7.

Table 4.7: Main rules of implementations descriptors

| |
|---|
| **An implementation is described by a set of descriptors.**<br>$descriptors : IMP \to P(Desc)$<br>$\forall (imp_i, imp_j) \in IMP, i \neq j \Rightarrow descriptors(imp_i) \cap descriptors(imp_j) = \emptyset$ |

*Example.* To execute a web service implementation, multiple descriptors are needed such as the URL (Uniform Resource Locator), the namespace and the operation name of the web service. To execute a Java implementation, other descriptors are needed such as the location of the Jar (Java archive) file where the program is defined, the name of the class containing the method to invoke and the name of the method to invoke.

### 4.4 Models

A model is an abstraction of concepts having the same characteristics. A model is an instance of a metamodel which plays the role of the modeling language or the modeling formalism. Relationships between models and metamodels are described in Table 4.8.

Table 4.8: Main rules of models

| |
|---|
| **A model is an instance of a unique metamodel.** |
| $MMinstOf : MM \rightarrow P(M)$ |
| $MMinstOf$ is a function that returns the set of instances of a metamodel. |
| $\forall(mm_i, mm_i) \in MM, i \neq j \Rightarrow MMinstOf(mm_i) \cap MMinstOf(mm_j) = \emptyset$ |

### 4.5 Instances

An instance represents data of the real world that are described by models. Relationships between models and instances are defined by the rules presented in Table 4.9.

Table 4.9: Main rules of instances

| |
|---|
| **A data instance is an instance of only one model.** |
| $MinstOf : M \rightarrow P(Inst)$ |
| $MinstOf$ is a function that returns the set of instances of a model. |
| $\forall(m_i, m_i) \in M, i \neq j \Rightarrow MinstOf(m_i) \cap MinstOf(m_j) = \emptyset$ |

Once the conceptual elements, that are necessary for metamodeling and model management, are defined, we need to extend the algebra of the PMMS exploitation language with operators for supporting the definition (i.e., create and delete) of operations and implementations and an operator permitting the execution of the defined operations. Next section gives the details of the extension of the algebra of the PMMS exploitation language.

## 5 Extension of the algebra of the PMMS exploitation language

As we have used the OntoDB/OntoQL PMMS, we propose to extend the algebra of its associated exploitation language defined in [Jean et al., 2007]. We notice that this algebra covers the main operators of PMMS exploitation language. Thus, our proposition can be applied to other PMMS exploitation languages. Our extension consists in 2 parts:

- the first one addresses the definition of creation and delete operators to create and delete operations and implementations;
- the second one focuses on the definition of the *RUN* operator for the exploitation of the defined operations. The purpose of defining such operator is to equip the model and data manipulation language with the capability to execute the user-defined operations. In other words, this operator is the interpreter associated to the defined operations.

Next subsections introduce respectively the creation, the delete and the RUN operators. These actions are defined by a signature (*SIG*), a precondition (*PRE*) and a postcondition (*POST*). We use the prime (') notation to describe the variables before (x) and after (x') operations executions.

## 5.1 Creation and delete operators

### 5.1.1 Creation actions

The main creation actions that a PMMS definition language should fulfill are defined below. The creation of an element implies, each time, the update of the concerned sets and the functions where an updated set is a domain and/or a range.

**Creation of a metamodel.**
SIG: $addMetaModel : String \rightarrow void$
*void* represents the *null* type
$addMetaModel(s)$
PRE: $\forall mm_i \in MM, s \neq mmName(mm_i)$
POST: $\exists mm$ such that: $MM' = MM \cup \{mm\} \wedge (mmName : MM' \rightarrow String) \wedge$
$(mmName(mm) = s) \wedge (classes : MM' \rightarrow P(CL)) \wedge (classes(mm) = \emptyset)$

The creation of a new metamodel implies the update of the $MM$ set and of the *mmName* and *classes* functions.

**Creation of a class of a metamodel.**
SIG: $addClass : MM$ x $String \rightarrow void$
$addClass(mm, s)$
PRE: $(mm \in MM) \wedge (\forall cl_i \in classes(mm), s \neq clName(cl_i))$
POST: $\exists cl$ such that: $CL' = CL \cup \{cl\} \wedge (clName : CL' \rightarrow String) \wedge (clName(cl) = s) \wedge$
$(superClass : CL' \twoheadrightarrow CL') \wedge$
$(inheritedAttributes : CL' \rightarrow P(ATT)) \wedge (inheritedAttributes(cl) = \emptyset)$
$(definedAttributes : CL' \rightarrow P(ATT)) \wedge (definedAttributes(cl) = \emptyset)$
$(attributes : CL' \rightarrow P(ATT)) \wedge (attributes(cl) = \emptyset)$

The creation of a new class of a metamodel implies the update of the $CL$ set and the *clName*, *superClass*, *inheritedAttributes*, *definedAttributes* and *attributes* functions. A newly created class has no super class and no defined attributes by default.

**Creation of an attribute of a class.**
SIG: $addAttribute : CL$ x $String$ x $DT \rightarrow void$

$addAttribute(cl, s, dt)$
PRE: $(cl \in CL) \wedge (\forall att_i \in definedAttributes(cl), s \neq attName(att_i)) \wedge (dt \in DT)$
POST: $\exists att$ such that: $ATT' = ATT \cup \{att\} \wedge (attName : ATT' \rightarrow String) \wedge$
$(attName(att) = s) \wedge (typeOf : ATT' \rightarrow DT) \wedge (typeOf(att) = dt) \wedge$
$(definedAttributes : CL \rightarrow P(ATT')) \wedge (definedAttributes'(cl) = definedAttributes(cl) \cup \{att\}) \wedge$
$(inheritedAttributes : CL \rightarrow P(ATT')) \wedge$
$(attributes : CL \rightarrow P(ATT')) \wedge (attributes'(cl) = attributes(cl) \cup \{att\})$

The creation of a new defined attribute of a class implies the update of the $ATT$ set and the $attName$, $typeOf$, $inheritedAttributes$, $definedAttributes$ and $attributes$ functions.

**Creation of an operation.**
SIG: $addOperation : String \text{ x } DT^n \text{ x } (DT \cup void) \rightarrow void$
$addOperation(s, (dti_1, dti_2, ..., dti_n), dto)$
PRE: $(\forall op_i \in OP, s \neq opName(op_i)) \wedge ((dti_1, dti_2, ..., dti_n) \in DT^n) \wedge (dto \in DT \cup void)$
POST: $\exists op$ such that: $OP' = OP \cup \{op\} \wedge (opName : OP' \rightarrow String) \wedge (opName(op) = s) \wedge$
$(implementations : OP' \rightarrow P(IMP)) \wedge (implementations(op) = \emptyset) \wedge$
$(input : OP' \text{ x } \mathbb{N}^+ \rightarrow DT) \wedge (\forall j \in 1..n, input(op, j) = dti_j) \wedge$
$(output : OP' \rightarrow DT \cup void) \wedge (output(op) = dto)$

The creation of a new operation extends the $OP$ set and updates the $opName$, $input$, $output$ and $implementations$ functions. When an operation is created, it has no implementation.

**Creation of an implementation of an operation.**
SIG: $addImplementation : OP \rightarrow void$
$addImplementation(op)$
PRE: $(op \in OP)$
POST: $\exists imp$ such that: $IMP' = IMP \cup \{imp\} \wedge$
$(implementations : OP \rightarrow P(IMP')) \wedge (implementations'(op) = implementations(op) \cup \{imp\})$
$(descriptors : IMP' \rightarrow P(Desc)) \wedge (descriptors(imp) = \emptyset)$

The definition of a new implementation of an operation leads to the extension of the $IMP$ set and the update of the $implementations$ and $descriptors$ functions.

### 5.1.2 Delete actions

The main delete actions that a PMMS definition language should fulfill are presented below. Similarly to the creation actions, the removal of an element involves the update of the concerned sets and functions. For readability, we update only a subset of functions for each action.

**Delete a metamodel.**
SIG: $deleteMetaModel : MM \rightarrow void$
$deleteMetaModel(mm)$
PRE: $(mm \in MM) \wedge (classes(mm) = \emptyset)$
POST: $(MM' = MM \backslash \{mm\})$

The removal of a metamodel implies the update of the $MM$ set.

**Delete a class of a metamodel.**
SIG: $deleteClass : MM \text{ x } CL \rightarrow void$
$deleteClass(mm, cl)$
PRE: $(mm \in MM) \wedge (cl \in classes(mm)) \wedge (definedAttributes(cl) = \emptyset)$
POST: $(CL' = CL\backslash\{cl\}) \wedge (classes : MM \rightarrow P(CL'))\wedge$
$(clName : CL' \rightarrow String) \wedge (superClass : CL' \twoheadrightarrow CL')\wedge$
$(ref : RefType \rightarrow CL')$

The delete of a class implies the update of the $CL$ set. All the associated functions are updated as well and all the subclasses of *cl* are deleted.

**Delete an attribute of a class.**
SIG: $deleteAttribute : CL \text{ x } ATT \rightarrow void$
$deleteAttribute(cl, att)$
PRE: $(cl \in CL) \wedge (att \in definedAttributes(cl))$
POST: $(ATT' = ATT\backslash\{att\}) \wedge (inheritedAttributes : CL \rightarrow P(ATT'))\wedge$
$(definedAttributes : CL \rightarrow P(ATT')) \wedge (attributes : CL \rightarrow P(ATT'))\wedge$
$(attName : ATT' \rightarrow String) \wedge (typeOf : ATT' \rightarrow DT)$

The deletion of an attribute implies the update of the $ATT$ set and the $attName$, $typeOf$ functions. Moreover, only the defined attributes can be deleted from a class as a class is not allowed to delete the inherited attributes which belong to its super class. Then, the deletion of a defined attribute leads to update the $definedAttributes$, $inheritedAttributes$ and $attributes$ functions.

**Delete an operation.**
SIG: $deleteOperation : OP \rightarrow void$
$deleteOperation(op)$
PRE: $(op \in OP) \wedge (implementations(op) = \emptyset)$
POST: $(OP' = OP\backslash\{op\}) \wedge (opName : OP' \rightarrow String)\wedge$
$(input : OP' \text{ x } \mathbb{N}^+ \rightarrow DT) \wedge (output : OP' \rightarrow DT \cup void)$

The removal of an operation implies the update of the $OP$ set and of the $opName$, $input$, $output$ functions.

**Delete an implementation of an operation.**
SIG: $deleteImplementation : OP \text{ x } IMP \rightarrow void$
$deleteImplementation(op, imp)$
PRE: $(op \in OP) \wedge (imp \in implementations(op))$
POST: $(IMP' = IMP\backslash\{imp\}) \wedge (Desc' = Desc\backslash descriptors(imp))\wedge$
$(implementations : OP \rightarrow P(IMP')) \wedge (descriptors : IMP' \rightarrow P(Desc'))$

The delete of an implementation leads to update of the $IMP$ and $Desc$ sets. When an implementation is deleted, all its corresponding descriptors are also deleted. Consequently, the $implementations$ and $descriptors$ functions are updated.

## 5.2 Run operator

Most PMMSs are equipped with a language whose algebra includes relational-like operators (e.g., projection or selection) for models and metamodels and built-in operators whose interpretation is fixed. The extension we propose makes possible to define new operations that could be executed. Therefore, there is a need to add, to the current algebra, a higher order operator capable to run model and data management operations given as parameters and whose implementation is defined. This algebra should be extended to be able to execute the operations that can be defined in our proposed extension. To fulfill this need, we define the *RUN* operator. We only give the signature of this operator in table 4.10 since its semantics depends on the processing defined in the implementation associated to the corresponding operation.

Table 4.10: Formalization of the RUN operator

| |
|---|
| SIG: $RUN : OP$ x $IMP$ x $(Inst \oplus M \oplus CL)^* \rightarrow (Inst \oplus M \oplus CL)$ |
| Where $\oplus$ represents the disjoint union |
| $RUN(op, imp, \{param_1, param_2, ..., param_n\}) = j$ |
| PRE: $(imp \in implementations(op)) \wedge (typeOf(param_i) = input(op, i), i \in 1..n)$ |
| POST: $typeOf(j) = output(op)$ |

*Example.* Let us consider the `PLIBClass2OWLClass` operation that transforms a PLIB class to an OWL one. This operation can be used, for instance, to transform the `Student` PLIB class (of the example presented in Chapter 1) to the `O_Student` OWL class. Thus, in this case, the *RUN* operator is invoked as follows:

$RUN(PLIBClass2OWLClass, PLIBClass2OWLClassImp, Student)$.

*Meaning.* The RUN operator takes as input the `PLIBClass2OWLClass` operation, an associated implementation (`PLIBClass2OWLClassImp`) and the `Student` PLIB class to transform. It returns the $O\_Student$ OWL class.

The RUN operator can operate at both model and data layers as it can execute operations of model management as well as operations of data management.

*Example.* If we consider that `age` is a derived property computed from the `birthday` property of the `Student` class, the invocation of the *RUN* operator to compute the age of the `Student1` instance is:

$RUN(computeAge, computeAgeImp, Student1)$.

*Meaning.* The RUN operator takes as input the `computeAge` operation, an implementation (`computeAgeImp`) and the `Student1` instance to which we compute the age property. The RUN operator returns the value corresponding to the age of the instance `Student1`.

## 6 Conclusion

In this chapter we have presented the formal model, using set theory, of the extension of PMMSs in order to integrate flexible behavioral semantics. This extension has addressed the PMMS data model which has been extended with new concepts to support the dynamic introduction of model management operations. Then, we have extended the logical metametamodel and the logical metamodel of the PMMS model repository in order to store the signatures of the defined operations and the descriptions of their associated implementations. Finally, our extension has focused on the algebra of the PMMS exploitation language which has been enriched with new operators to enable the definition and the exploitation of model and data management operations. Our approach is implemented on a relational database but can as well be implemented on other types of DBMSs provided that the PMMS API is implemented for the targeted DBMS.

Next chapter presents the extension of the OntoQL grammar with instructions for the definition and the exploitation of model management operations. Besides, it introduces the prototyping of our approach.

Chapter 5

# BeMoRe : extension of the exploitation language and prototyping

**Contents**

**Abstract.** In the previous chapter we have presented the formal extension of PMMSs to support behavioral semantics. This chapter introduces the extension of the OntoQL grammar with the capability to define and exploit operations and implementations. Moreover, it presents the BeMoRe prototype and a preliminary performance study of our prototype in order to show the scaling of our approach.

## 1 Introduction

In Chapter 4 we have presented the generic and formal extension of PMMSs in order to handle behavioral semantics. This concerns the extension of the PMMS metametamodel with new concepts in order to support operations that can be implemented using flexible mechanisms such as stored procedures, external programs and web services. Furthermore, we have presented the logical extension of the PMMS model repository in order to store signatures of operations and their associated implementations. Then, we have introduced the extension of the algebra of the PMMS exploitation language with new operators to create, manipulate and exploit operations.

This chapter presents the extension of the OntoQL syntax and the BeMoRe prototype. The implementation of our approach under the OntoQL language is presented through two parts. The first part is dedicated to the extension of OntoQL in order to support model management operations operating at the model level and addressing model elements. The second part is devoted to the extension of the OntoQL to support data management operations operating at the data level. This kind of operations can be useful for instance for data migration. Nonetheless, the extension of the PMMS model repository and its exploitation language is not sufficient since we need a mechanism to run external program and web services for the PMMS. To do so, we have set up an API, called the *behavior API*, which is a part of the BeMoRe prototype and which we present in this chapter.

The remainder of this chapter is as follows. Section 2 introduces the extension of the OntoQL syntax to handle model and data management operations. Section 3 exposes the BeMoRe prototype. Section 4 presents a small performance evaluation of the BeMoRe prototype. Finally, Section 4 concludes this chapter.

## 2 Extension of the OntoQL language

### 2.1 Extension with model management operations

This section presents the extension of the OntoQL language with the capability to define, manipulate and exploit operations and their implementations. The extension made on the OntoQL language can be decomposed into 2 parts:

- the first part concerns the extension of OntoQL with the *Model Behavior Definition Language* (MBDL) for the definition of operations and implementations. It is an extension of the OntoQL language with new statements to create and delete operations and implementations; and associate implementations to the operations they implement. Furthermore, we have extended OntoQL with the capability to choose a default implementation in the case where multiple implementations are available for an operation. Another feature we have introduced covers the possibility to explicit, in an OntoQL statement, the implementation to execute for an invoked operation;

- the second part is related to the extension of the *Model Manipulation Language* of OntoQL to include the capability to invoke operations in OntoQL statements. This consists in the capability to make operations calls in OntoQL queries.

Each of these two parts is detailed below.

### 2.1.1 Model behavior definition language

#### Create Operation Statement

We have extended the OntoQL grammar with the capability to create operations at the metamodel level. These operations are dedicated to operate at the model level and address models elements. Each operation is characterized with a name and has a set of inputs and an output data types as it is defined in the following grammar. The syntax to create a model management operation is given below.

```
<m2 operation definition> ::= CREATE OPERATION <operation name> [<descriptors clause>]
                              [<input clause>] [<output clause>]
<descriptors clause>      ::= DESCRIPTOR (<descriptors list>)
<descriptor>              ::= <descriptor name> = <descriptor value>
<input clause>            ::= INPUT (<datatype list>)
<output clause>           ::= OUTPUT (<datatype>)
```

*Example.* The syntax to create the PLIBClass2OWLClass operation is given by Listing 5.1.

Listing 5.1: Statement for creating the PLIBClass2OWLClass operation

```
CREATE OPERATION #PLIBClass2OWLClass
INPUT (REF (#PLIBClass))
OUTPUT (REF (#OWLClass));
```

*Meaning.* This OntoQL statement defines a model management operation prefixed with the # character to distinguish model management operations and data management as in OntoQL models and instance elements are differentiated by the # character. The defined operation has as input a PLIBClass and returns an OWLClass. The keyword REF indicates that the data type is a complex type that references a metamodel element.

#### Create Implementation Statement

Once an operation is defined, one or many associated implementations can be defined. Yet, knowing that heterogeneous types of implementations may be set up (e.g., external programs, web services), these implementations are described by different descriptors which are metadata of programs that are necessary for the execution of implementations. For instance, to execute a Java external program, we need to know the location of the Java archive (JAR) file where the implementation is defined, the class name and the method name. And to execute a web service, we need to know the URL[3] of the web service and the web service operation name. Thus, the OntoQL instruction to create an implementation should be generic in the sense that it shall support the creation of any type of implementation that can be described using different descriptors. Thus, as presented in the previous chapter, an implementation is characterized by a name and a set of descriptors represented by pairs of (*key*, *value*). The statement for creating an implementation should specify the operation that is implemented. Thus, the OntoQL syntax to create an implementation at the metamodel level is given below.

[3]Unified Resource Locator

```
<m2 implementation definition> ::= CREATE IMPLEMENTATION <mimplementation name>
                                   <descriptors clause> <implements clause>
<implements clause>            ::= IMPLEMENTS <moperation or operation name>
```

*Example.* Listing 5.2 statement creates an implementation of the `PLIBClass2OWLClass` operation.

Listing 5.2: Statement for creating an implementation of `PLIBClass2OWLClass`

```
CREATE IMPLEMENTATION #PLIBClass2OWLClassImp
DESCRIPTORS (type = 'java',
             location = 'http://.../programs.jar',
             class = 'fr.ensma.lias.myClass',
             method = 'PLIBClass2OWLClass')
IMPLEMENTS #PLIBClass2OWLClass;
```

*Meaning.* this statement creates an implementation of the `PLIBClass2OWLClass` operation (`PLIB-Class2OWLClassImp`). It provides descriptors of a Java program stored outside the database. In particular, these descriptors specify the file location of the external program, the Java class where the method is defined and the method to execute. Besides, the create statement indicates the implemented operation (`PLIBClass2OWLClass`).

### Delete Operation Statement

We have also extended the OntoQL grammar with instructions permitting to delete operations and implementations. The syntax for deleting an operation is defined as follows.

```
<m2 operation removal> ::= DELETE OPERATION <operation name>
```

*Example.* The syntax to delete the `PLIBClass2OWLClass` operation is defined by Listing 5.3.

Listing 5.3: Statement for deleting the `PLIBClass2OWLClass` operation

```
DELETE OPERATION #PLIBClass2OWLClass
```

*Note.* When a model management operation is deleted, its associated implementations are also removed.

### Delete Implementation Statement

The syntax for deleting an implementation is given below.

```
<m2 implementation removal> ::= DELETE IMPLEMENTATION <implementation name>
```

*Example.* The syntax to delete the `PLIBClass2OWLClassImp` implementation is given by Listing 5.4.

Listing 5.4: Statement for deleting the `PLIBClass2OWLClassImp` implementation

```
DELETE IMPLEMENTATION #PLIBClass2OWLClassImp
```

*Note.* If an implementation of an operation is removed, the operation can use other implementations if they exist.

### Set Default Implementation Statement

If an operation is associated to several implementations, the implementation to execute is chosen randomly by the behavior API. However, the execution of some implementations may be faster than the execution of other ones, or some implementations may be not available, etc. Thus, the OntoQL language has also been extended with the possibility to define a default implementation. If several implementations are available for an operation, the default implementation is used for the execution of the operation. The syntax to define a default implementation for an operation is given below.

```
<set default implementation definition>  ::= SET DEFAULT IMPLEMENTATION
                                                 <implementation name> <for operation clause>
<for operation clause>                   ::= FOR <operation name>
```

This syntax is the same to set the default implementation for model management and data management operations.

*Example.* Listing 5.5 statement defines a default implementation for the `PLIBClass2OWLClass` operation.

Listing 5.5: Set a default implementation for a model management operation

```
SET DEFAULT IMPLEMENTATION #PLIBClass2OWLClassImp
FOR #PLIBClass2OWLClass;
```

*Meaning.* The first statement defines `PLIBClass2OWLClassImp` as the default implementation for the `PLIBClass2OWLClass` operation.

### Assigning a specific implementation

If, for instance, the default implementation is not available, the execution of the invoked operations cannot be achieved. Thus, the solution is to change the default implementation and execute again the query. To avoid this kind of problems, we have extended OntoQL with the capability to assign a specific implementation to be executed for an operation directly in an OntoQL select statement. The associated syntax is defined below.

```
<select statement>            ::= <select clause> <from clause> <where clause> ...
                                  <using implementation clause>
<using implementation clause> ::= USING IMPLEMENTATION <implementation operation list>
<implementation operation>    ::= <implementation name> -> <operation name>
```

*Example.* Listing 5.6 shows an example of this behavior.

Listing 5.6: Specifying the implementation to run in a select statement

```
CREATE #OWLClass O_Student AS
   SELECT #PLIBClass2OWLClass(c)
   FROM #PLIBClass AS c
   WHERE c.#name = Student
   USING IMPLEMENTATION #PLIBClass2OWLClassImp->#PLIBClass2OWLClass;
```

*Meaning.* This statement creates an OWL class (`O_Student`) by transforming the `Student` PLIB class using the `PLIBClass2OWLClass` operation. The `USING IMPLEMENTATION` clause indicates explicitly that the implementation to be executed for the `PLIBClass2OWLClass` operation is `PLIBClass-2OWLClassImp`.

The `USING IMPLEMENTATION` clause is helpful only when multiple implementations are available for an operation. In the case where it is not present, either the assigned implementation or the default implementation is run.

### 2.1.2 Extension of the model manipulation language

When an operation and at least one associated implementation are defined, the operation can be invoked in an OntoQL select statement. Thus, the OntoQL Model Manipulation Language (MML) has been extended in order to make possible operations invocations in OntoQL select statements. Listing 5.7 shows an example of such a behavior.

Listing 5.7: Example of an operation invocation

```
CREATE #OWLClass O_Student AS
   SELECT #PLIBClass2OWLClass(c)
   FROM #PLIBClass AS c
   WHERE c.#name = 'Student';
```

*Meaning.* This statement shows the creation of a derived OWL class (`O_Student`) from an existing PLIB class (`Student`).

Next section presents the extension of the OntoQL language with data management operations.

## 2.2 Extension with data management operations

Similarly to the extension presented in the last section, the extension of the model level of the OntoDB/OntoQL PMMS consists in extending firstly the model layer of the OntoDB architecture before extending the OntoQL language. Next subsections expose both extensions.

### 2.2.1 Instance behavior definition language

#### Create Operation Statement

We have equipped OntoQL with the capability to create operations at the model level. These operations are dedicated to act at the instance level and relate to instance elements only. Each operation is characterized by a name and has a set of inputs and an output as defined in the following BNF rule.

```
<m1 operation definition> ::= CREATE #OPERATION <operation name> [<descriptors clause>]
                              [<input clause>] [<output clause>]
```

*Example.* Listing 5.8 statement creates the `computeAge` operation defined at the model level.

Listing 5.8: Statement for creating the `computeAge` operation

```
CREATE #OPERATION computeAge
INPUT (REF (Person))
OUTPUT (INTEGER);
```

*Meaning.* This statement creates an operation which computes the age of a person. This operation has as a parameter the `Person` class and returns an integer corresponding to the type of the age property.

*Note.* We differentiate operations of the metamodel level and those of the model level with the # character. This is conform to the OntoQL grammar which separates concepts of the different model layers i.e., concepts of the metamodel and the model layers.

#### Create Implementation Statement

The BNF rule defining the creation of an implementation at the model level is described as follows.

```
<m1 implementation definition> ::= CREATE #IMPLEMENTATION <implementation name>
                                   <descriptors clause> <implements clause>
```

*Example.* Listing 5.9 statement creates an implementation of the `computeAge` operation.

Listing 5.9: Statement for creating an implementation of `computeAge`

```
CREATE #IMPLEMENTATION computeAgeImp
DESCRIPTORS (type = 'java',
             location = 'http://.../programs.jar',
             class = 'fr.ensma.lias.myClass',
             method = 'computeAge')
IMPLEMENTS computeAge;
```

*Meaning.* The previous statement creates an implementation of the `computeAge` operation. This implementation is a Java program characterized by 4 descriptors: `type` determines the type of the implementation which is Java in this case, `location` indicates the location of the Java archive file containing the Java program, `class` represents the class name where the Java method is defined and `method` defines the method name to run.

### Delete Operation Statement

The BNF rule for deleting a data management operation is defined as follows.

```
<m1 operation removal> ::= DELETE #OPERATION <operation name>
```

*Example.* Listing 5.10 statement deletes the `computeAge` operation.

Listing 5.10: Statement for deleting the `ComputeAge` operation

```
DELETE #OPERATION ComputeAge
```

*Note.* Similarly to model management operations, when a data management operation is deleted, its associated implementations are implicitly deleted as well.

### Delete Implementation Statement

The BNF rule for deleting an implementation of the model level is:

```
<m1 implementation removal> ::= DELETE #IMPLEMENTATION <implementation name>
```

*Example.* Listing 5.11 statement deletes the `computeAgeImp` implementation.

Listing 5.11: Statement for deleting the `computeAgeImp` implementation

```
DELETE #IMPLEMENTATION ComputeAgeImp
```

*Note.* Similarly to implementations for model management, if the default implementation of a data management operation is removed, the operation can use other implementations if they exist.

### Set Default Implementation Statement

OntoQL language is also extended with the possibility to define a default implementation if several implementations are available for a data management operation. The syntax to define a default implementation for a data management operation is similar to the one related to model management operations.

*Example.* Listing 5.12 statement defines a default implementation for the `computeAge` operation.

Listing 5.12: Set a default implementation for a data management operation

```
SET DEFAULT IMPLEMENTATION computeAgeImp
FOR computeAge;
```

*Meaning.* This statement defines the `computeAgeImp` as the default implementation of the `computeAge` operation.

#### Assigning a specific implementation

We can also assign a specific implementation to be executed for an operation directly in an OntoQL select statement. The associated syntax is similar to the one addressed for model management operations.

*Example.* Listing 5.13 shows an example of this behavior.

Listing 5.13: Specifying the implementation to run in a select statement

```
SELECT computeAge(P)
FROM Person AS P
WHERE P.name = 'Dupond'
USING IMPLEMENTATION computeAgeImp->computeAge;
```

*Meaning.* This statement computes the age of a person p whose name is 'Dupond'. The `USING IMPLEMENTATION` clause indicates explicitly that the implementation to be executed for the `computeAge` operation is `computeAgeImp`.

### 2.2.2 Extension of the data manipulation language

When a data management operation and at least one associated implementation are defined, the operation can be invoked in an OntoQL select statement of the data manipulation language (DML). Thus, the OntoQL DML has been extended in order to make possible operations invocations in OntoQL select statements. Listing 5.14 shows an example of data management operation invocation.

Listing 5.14: Example of an operation invocation

```
SELECT computeAge(P)
FROM Person AS P
WHERE P.name = 'Dupond'
```

*Meaning.* This statement computes a derived property. It computes the `age` property of the `Person` class using the `computeAge` operation.

In the next section, we present the general architecture of the BeMoRe prototype including the behavior API we have set up. Then, we describe the execution process of OntoQL statements taking into account operations invocations. At the end, we introduce a preliminary performance evaluation of the BeMoRe prototype to show the scaling of our approach.

## 3 The BeMoRe prototype

The BeMoRe prototype is built on top of the OntoDB/OntoQL prototype. Indeed, BeMoRe enriches OntoDB/OntoQL with the capability to define and exploit model and data management operations. Particularly, the logical extension of the OntoDB equips the model repository with tables to store signatures of operations and metadata for the description of their associated implementations. Moreover, the extension of the OntoQL grammar adds new instructions to define (create and delete) operations and implementations, and to support operations invocations in OntoQL statements. Indeed, the support of operations invocations within OntoQL statements (1) modifies the execution process of OntoQL queries as we have to take into account the possibility to have operations invocations in OntoQL statements, and (2) adds to the OntoDB/OntoQL prototype a new component which is the *behavior API*. This API bridges the OntoDB/OntoQL environment and the external environments (e.g., web services, external programs). Specifically, this API makes data types correspondences (e.g., using XML file) between OntoDB/OntoQL data types and those of external environments, and invokes the external programs and/or the remote services.

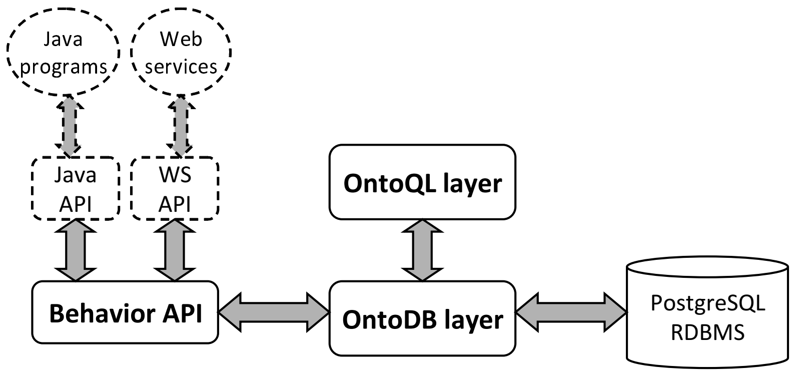


Figure 5.1: The architecture of the BeMoRe prototype

Figure 5.1 presents the general architecture of the BeMoRe prototype. This prototype is composed of 3 main layers: the OntoDB layer, the OntoQL layer and the behavior API layer.

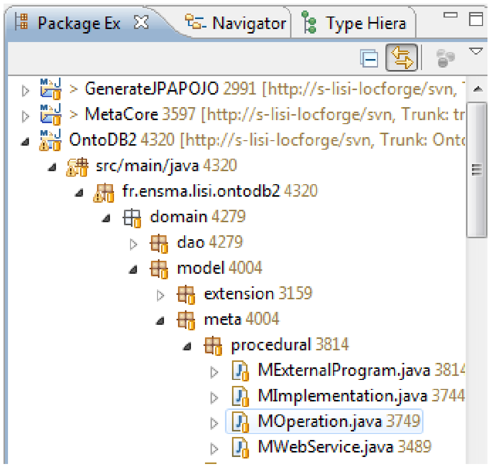
## 3.1 The OntoDB layer

The OntoDB layer is implemented on top of the PostgreSQL DBMS. It defines an API to store and manipulate metamodels, models and data in the PostgreSQL DBMS. The OntoDB layer defines primitives to create, update, query and delete metamodels, models and instances using Entreprise JavaBeans (EJBs). Indeed, an EJB corresponds to an entity of the PMMS metametamodel and maps the entity to a table in the database using the Java Persistence API (JPA). JPA provides primitives to persist objects in relational databases. Moreover, it offers a query language called JPQL (Java Persistence Query Language) to retrieve the persisted objects.

**Extension with model management operations**

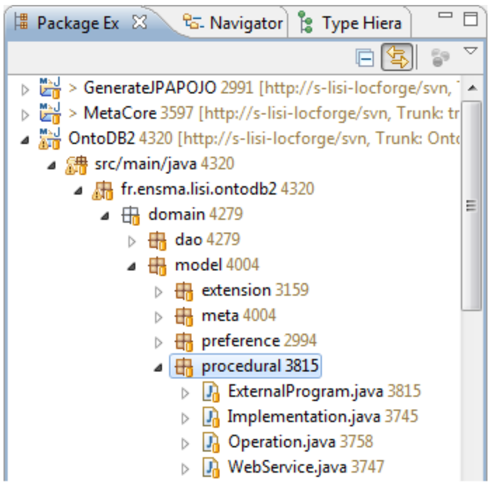
To be able to define and persist the signatures of model management operations and the descriptions of their associated implementations, we have extended the OntoDB metametamodel package with the `procedural` package (see Figure 5.2).

The added EJBs classes (`MOperation`, `MImplementation`) correspond respectively to the concepts of operation and implementation. Each EJB is mapped to a table using JPA to persist operations and implementations.



```
> GenerateJPAPOJO 2991 [http://s-lisi-locforge/svn,
> MetaCore 3597 [http://s-lisi-locforge/svn, Trunk: tr
OntoDB2 4320 [http://s-lisi-locforge/svn, Trunk: Onto
  src/main/java 4320
    fr.ensma.lisi.ontodb2 4320
      domain 4279
        dao 4279
        model 4004
          extension 3159
          meta 4004
            procedural 3814
              MExternalProgram.java 3814
              MImplementation.java 3744
              MOperation.java 3749
              MWebService.java 3489
```

Figure 5.2: Extension of the OntoDB metamodel package with the procedural package



```
> GenerateJPAPOJO 2991 [http://s-lisi-locforge/svn,
> MetaCore 3597 [http://s-lisi-locforge/svn, Trunk: tr
OntoDB2 4320 [http://s-lisi-locforge/svn, Trunk: Onto
  src/main/java 4320
    fr.ensma.lisi.ontodb2 4320
      domain 4279
        dao 4279
        model 4004
          extension 3159
          meta 4004
          preference 2994
          procedural 3815
            ExternalProgram.java 3815
            Implementation.java 3745
            Operation.java 3758
            WebService.java 3747
```

Figure 5.3: Extension of the OntoDB model package with the procedural package

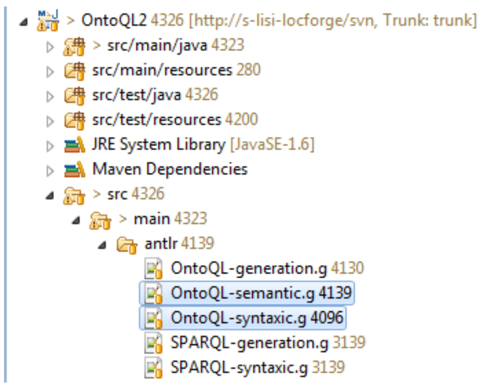
### Extension with data management operations

The extension of OntoDB with data management operations is similar to the extension of OntoDB with model management operations. Thus, in order to store the signatures of data management operations and the descriptions of their associated implementations, we have extended the OntoDB metamodel package with the `procedural` package (see Figure 5.3). The `procedural` package defines EJBs which play the role of operation and implementation.

## 3.2 The OntoQL layer

The OntoQL layer is an API that defines the syntactic and the semantic rules of OntoQL queries. Besides, it processes the syntactic and semantic analysis of the executed OntoQL queries. Then, an OntoQL query is translated to a native query (e.g., SQL, JPQL) and transmitted to the OntoQL interpreter to be executed.

At this stage, OntoQL language supports definition and querying of structural elements of metamod-



```
> OntoQL2 4326 [http://s-lisi-locforge/svn, Trunk: trunk]
  > src/main/java 4323
  src/main/resources 280
  src/test/java 4326
  src/test/resources 4200
  JRE System Library [JavaSE-1.6]
  Maven Dependencies
  > src 4326
    > main 4323
      antlr 4139
        OntoQL-generation.g 4130
        OntoQL-semantic.g 4139
        OntoQL-syntaxic.g 4096
        SPARQL-generation.g 3139
        SPARQL-syntaxic.g 3139
```

Figure 5.4: Grammar files of OntoQL

els and models besides of supporting querying of instances (Select clause). Our objective is to be able to define operations and implementations and to make operations invocations in OntoQL statements. Yet, to support these capabilities, we have extended the OntoQL grammar to support these capabilities. In particular, we extended the `OntoQL-syntaxic` and `OntoQL-semantic` ANTLR[4] grammar files shown in Figure 5.4. The `OntoQL-syntaxic` file defines the syntactic rules of OntoQL queries. For instance, a part of the syntax to support the creation of model management operations is given in Listing 5.15.

Listing 5.15: Example of the extension of the OntoQL syntax

```
createStatement
      : CREATE^ (entityDefinition | mBehaviorDefinition | ...)
      ;

mBehaviorDefinition
      : mOperationDefinition
      | mImplementationDefinition
      ;

mOperationDefinition
      : OPERATION^ identifier (inputClause)? (outputClause)?
      ;
```

*Meaning.* The first clause defines the create statement of the OntoQL language (`createStatement`). Among the possible choices of this clause, we find `mBehaviorDefinition` which represents the definition of a model management operation (`mOperationDefinition`) or an implementation of a model management operation (`mImplementationDefinition`). The definition clause of a model management operation clause is characterized by the keyword `OPERATION`, the identifier (the name) of the operation, its input and output data types.

The syntactic extension of OntoQL is not sufficient to precise the semantics of OntoQL statements creating model management operations and implementations. Thus, we have extended `OntoQL-semantic` with new clauses in order to retrieve different concepts when creating a model management operation. The semantic extension generates a tree, corresponding to the query, from which different concepts can be distinguished. For instance, an OntoQL query to create an operation is transformed by the compiler

[4]http://www.antlr.org/

to a tree from which the operation name, its input and output data types can be retrieved. A part of the semantic extension of the OntoQL grammar for creating operations is given in Listing 5.16.

Listing 5.16: Example of the extension of the OntoQL semantic

```
mBehaviorDefinition
    : mOperationDefinition
    | mImplementationDefinition
    ;

mOperationDefinition
    : #(OPERATION identifier (inputClause)? (outputClause)?)
    ;
```

The proposition made in this thesis faces three technical challenges:

- the first one is related to the capability of PMMSs to invoke external programs and web services since current PMMSs did not address this aspect. This challenge leads to the second one;
- the second challenge concerns the data exchange format between PMMSs and the external environments (i.e., web services and external programs). Indeed, data types of the BeMoRe prototype and those of the external environments are not similar. Thus, data types mappings are required;
- the third challenge relates to the process that can be followed to answer OntoQL queries containing invocations of the defined operations.

To meet these challenges, we have set up an API, that we have called *behavior API*, which fulfills these needs. This API is presented in the next section.

## 3.3 The behavior API layer

The behavior API defines a new application programming interface (API), called *Behavior API*, which plays the role of the bridge between the OntoDB/OntoQL environment and the external environments. Indeed, this API makes data types correspondences between data types of the OntoDB/OntoQL PMMS and data types of the different implementation types. Moreover, the behavior API has been implemented to run external programs and remote services. The behavior API defines primitives for data types mappings and the execution of any type of implementation (e.g., web services, Java, Ada). Indeed, the integration of a new type of implementation requires the implementation of (1) the primitive for data types correspondences between OntoDB/OntoQL data types and the data types of the new implementation, and (2) the implementation of the primitive to execute the remote program and to return the result.

## 3.4 Handling operations invocations in OntoQL

When an OntoQL query is executed, the process to run this query follows different steps (Figure 5.5).

- *Syntactic analysis*: the OntoQL API processes at first the syntactic analysis to check if the query respects the OntoQL grammar. If the step of the syntactic analysis is passed, the query goes through a semantic analysis.

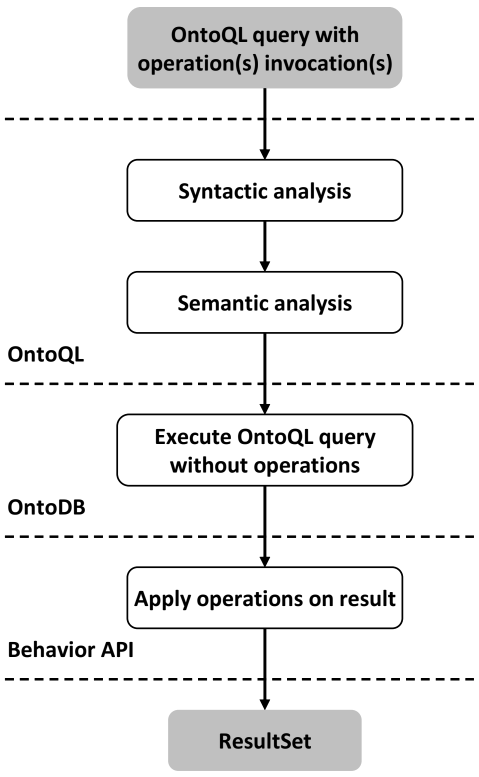

OntoQL query with operation(s) invocation(s)
Syntactic analysis
Semantic analysis
OntoQL
Execute OntoQL query without operations
OntoDB
Apply operations on result
Behavior API
ResultSet

Figure 5.5: Execution process of OntoQL statements containing operations invocations

- *Semantic analysis*: at this step, the OntoQL layer verifies, for instance, if the queried classes exist in the PMMS, or if the invocation of an operation conforms to its definition, etc .Then, the OntoQL query is translated to a native query (e.g., SQL, JPQL) omitting operations. This allows to get all the data before applying operations on these data.
- *Native query execution*: the OntoDB layer executes the native query provided by the OntoQL layer and returns the corresponding result set.
- *Operations invocations*: when the OntoDB layer gets the result of the execution of the native query, the data are transmitted to the behavior API in order to process operations on these data. Thus, the behavior run the external program(s) and/or the web service(s) using the data result set provided by the execution of generated native query. Next, the behavior API returns the result to the OntoDB layer which stores the result of the initial OntoQL query in the database.

To illustrate the execution process of OntoQL queries containing operations invocations, we explain how the OntoQL statement of Listing 5.17 is answered.

Listing 5.17: An OntoQL with an operation invocation

```
SELECT name, computeAge(birthday)
FROM Student
```

If the syntactic analysis of the query is successful, this query is translated to the SQL statement of Listing 5.18 omitting the invocation of the `computeAge` operation.

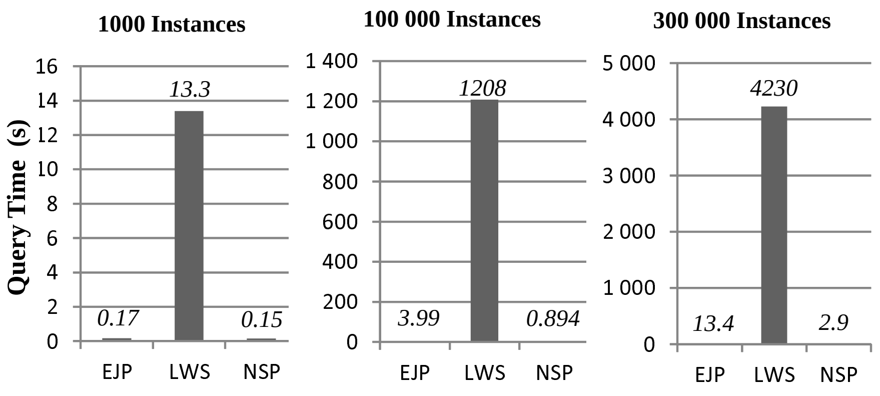


Figure 5.6: Performance evaluation of BeMoRe

Listing 5.18: The resulting SQL query

```
SELECT name, birthday
FROM Student
```

Then, the OntoQL layer transmits to the OntoDB layer (1) the generated native query (of Listing 5.18), (2) the invoked operation (`computeAge`) and (3) and indicates that the operation is applied to the `birthday` property. Next, the OntoDB layer executes the generated SQL query and transmits to the behavior API the result set of the SQL query and the metadata of the `computeAge` implementation. Finally, the behavior API runs the external implementation of the invoked operation and returns the results to the OntoDB layer which persists the final result in the OntoDB repository. Next section presents a preliminary performance evaluation of the BeMoRe prototype.

## 4 A preliminary performance evaluation

As a first step to study the scalability of our implementation we compare the execution time of the model query of Listing 5.19 (similar results were obtained for a metamodel query).

Listing 5.19: Queries used for our experimentations

```
SELECT computeAge(s)
FROM Student AS s
```

We execute these queries using three types of implementations of the *computeAge* function: native stored procedure (*NSP*), external Java program (*EJP*) and local web service (*LWS*) on three different sizes of data (1000, 100000 and 300000 instances). These experiments were run on the OntoDB/OntoQL PMMS based on PostgreSQL 8.2 installed on a standard Intel Core Duo E6550 2.33 Ghz 3GB of RAM desktop machine.

The performance numbers for the query on the three data sizes and for the three implementations are shown in Figure 5.6. All times presented (in seconds) are the average of three runs of the queries.

As expected the invocation of NSP performs a factor of 4-5 faster than EJP and largely faster than LWS. As the EJP and LWS are called one time for each instance, the time of queries increases nearly linearly with the size of data. To optimize this process, this result suggests to design a Java method or a Web services that takes as input a set of data instead of an individual data. A more complete study of the problem of query optimization for PMMSs is part of our future work.

## 5 Conclusion

In this chapter we have presented our modification of the syntactic part of the OntoQL language in order to support behavioral semantics. Our extension concerns both the metamodel and the model parts of the OntoQL language in order to support model management and data management operations. Multiple implementations can be called in the same statement using the `USING IMPLEMENTATION` clause. Moreover, as OntoQL can be used to query both the metamodel and model levels, operations at both levels can be combined in a query. Furthermore, we have exposed the technical aspects of the BeMoRe prototype, an extension of the OntoDB/OntoQL prototype. Our prototype extends the OntoDB layer with structures (tables) to store operations and implementations, and extends the OntoQL language with statements to create and exploit operations for model and data management. We have also shown that this extension modifies the architecture of the OntoDB/OntoQL prototype since the architecture of BeMoRe includes an API, called behavior API, that makes the bridge between the OntoDB/OntoQL environment and the external environment. This API makes data types correspondences between the two environments and runs external programs and remote services. Moreover, we have presented our extension of the OntoQL query language with the capability to invoke operations in OntoQL queries. This extension modifies the execution process of the OntoQL statement. The contribution presented in this part of our thesis has been validated in [Bazhar et al., 2013a].

The next part of this thesis presents use cases of our approach which consists in extending PMMSs with behavioral semantics. These use cases concern the usage of our proposition to compute derived ontologies concepts and its usage to enhance a methodology to design databases storing ontologies and data. Finally, the last use case addresses model transformation and model analysis using the proposition we have made in this thesis.

# Part III

# Applications

Chapter 6

# Managing non canonical concepts in ontology-based databases

**Contents**

**Abstract.** In the previous part of this thesis, we have presented the different concepts of our proposition to extend PMMSs with behavioral semantics. To show the usefulness of this proposition, we apply our approach to a first case study related to *Ontology-Based DataBases (OBDBs)* which are specific PMMSs dedicated to store and manipulate ontologies (conceptual models shared over large communities) together with the data they describe. If existing OBDBs support canonical (primitive) concepts, they do not address sufficiently *non canonical concepts* (derived or defined concepts). Indeed, existing OBDBs provide specific and hard-coded mechanisms to compute these non canonical concepts. In

this chapter we show how our approach is used to support non canonical concepts in OB-DBs in a dynamic and flexible way. As a running example, we consider the support of non canonical constructors of the Ontology Web Language (OWL) in the OntoDB platform. This application has been validated in [Bazhar et al., 2012a].

## 1 Introduction

Since its introduction, the notion of ontology has been widely used and has regained a lot of interests with the recent development of the Semantic Web. Indeed, ontologies are defined in a lot of domains such as engineering, medicine, biology or chemistry and are set up for a wide range of applications like natural language processing, information retrieval, electronic commerce, software component specification or information systems integration.

The intensive use of ontologies in a large number of domains leads to two main difficulties. (1) The amount of data described by ontologies can be huge, especially in domains like e-commerce, engineering or Semantic Web. (2) Diverse ontology formalisms (ontology models) exist to define different types of ontologies. For example, a lot of ontologies in engineering are defined with the PLIB (Parts LIBrary) ontology language [Pierra, 2007], [Pierra and Sardet, 2010] whereas most ontologies in the Semantic Web are defined with languages such as RDF Schema (or RDFS) [Brickley and Guha, 2004] or OWL (Ontology Web Language) [Dean and Schreiber, 2004]. The first difficulty has been overcome with the introduction of a new type of databases, called *Ontology-Based DataBases (OBDBs)*, that store both data and the ontologies which define the meaning of these data. An OBDB is a specific PMMS dedicated to store and manage ontologies and their data instances. Several OBDBs have been proposed in the literature (e.g., *Jena* [Carroll et al., 2004], *Sesame* [Broekstra et al., 2002], *Oracle* [Chong et al., 2005], *RStar* [Lu et al., 2007]). The second issue (the wide diversity of existing ontologies) has been overcome thanks to metamodeling capabilities provided by some OBDBs. For example, OntoDB [Dehainsala et al., 2007] provides the capacity to introduce and support multiple ontology formalisms, and thus the storage and the manipulation of ontologies expressed with different formalisms.

However, OBDBs remain incomplete since they do not address sufficiently *non canonical concepts* (derived concepts) i.e., concepts defined by a complete axiomatic definition expressed in terms of other concepts [Gruber, 1995]. This kind of concepts are particularly important in the OWL ontology model. For example, with this ontology formalism, non canonical classes can be defined as union of other classes or as a restriction on a property value, etc. As existing OBDBs are usually defined to store ontologies expressed with specific formalisms, they handle non canonical concepts using hard-coded mechanisms (e.g., internal reasoners, predefined operators) which are not always suitable for computing non canonical concepts of all types of ontologies.

Our claim is that non canonical concepts (NCCs) can be handled in a dynamic and flexible way in an OBDB. This goes by exploiting our proposition, presented in the previous chapters, to extend PMMSs dynamically with model management operations that can be implemented using internal mechanisms (e.g., database procedural language, predefined operators, internal reasoners) as well as external ones such as web services, external reasoners, external programs (e.g., Java programs), etc. The work achieved in this chapter has been validated in [Bazhar et al., 2012a].

In this chapter, we consider the OntoDB OBDB [Dehainsala et al., 2007]. This OBDB has originally been defined for the PLIB ontology model but can support other ontology formalisms thanks to the metamodeling capabilities it provides. However, OntoDB did not provide, before our proposition, flexible mechanisms to support the definition of non canonical concepts. Thus, we show in this chapter how the extension made under the OntoDB/OntoQL platform can be exploited to compute derived ontologies concepts. As a proof of concepts, we show how the non canonical constructors of the OWL language can be defined and implemented on OntoDB.

The remainder of this chapter is organized as follows. Section 2 introduces a background on the notion of ontology. Section 3 presents a state of the art of OBDBs and shows their limitations regarding the support of non canonical concepts. Section 4 presents the support of structural concepts of the OWL formalism in the OntoDB/OntoQL platform. Section 5 exposes the support the non canonical constructors of OWL using our approach. Finally, Section 6 concludes this chapter.

## 2 Background on ontologies

Ontologies are used in different domains like the Semantic Web, engineering and e-commerce, and are set up for several types of applications like systems integration, databases, etc. Various definitions of an ontology have been proposed in the literature but the most used one is the definition of Gruber *an explicit specification of a conceptualization* [Gruber, 1993].

Different languages have been proposed to design ontologies. These languages are often devoted to design ontologies of a specific domain. For instance, OWL (Ontology Web Language) [Dean and Schreiber, 2004], RDF (Resource Description Framework) [Manola and Miller, 2004] and RDF Schema [Brickley and Guha, 2004] are dedicated to design ontologies of the Semantic Web, while PLIB (Parts LIBrary) [Guy et al., 2003] is devoted to design ontologies in engineering especially in mechanics.

### 2.1 Characteristics of an ontology

An ontology is characterized by the 3 following features.

- *Formal*: an ontology is defined using a formal language that provides the capability to express logical axioms. This makes possible automatic processing on ontologies (e.g., reasoning, checking the conformity of concepts).
- *Consensual*: an ontology is accepted by and shared over a large community in a domain.
- *Referenced*: each concept of an ontology has a unique identifier to identify this concept whatever is the formalism used to design the ontology. This aspect is useful for the interoperability of the ontology which eases tasks such as systems integration and data exchange.

### 2.2 Taxonomy of ontologies

Different types of ontologies exist, and 3 types are distinguished as represented in the Figure 6.1 [Jean et al., 2006c]:

- *Canonical ontologies (COs)*: define primitive concepts i.e., primitive classes and properties. These primitive concepts cannot be computed nor derived from other concepts. For instance, the example of Figure 6.1 defines a canonical class (`Student`) and a canonical property (`gender`) at the CO layer.

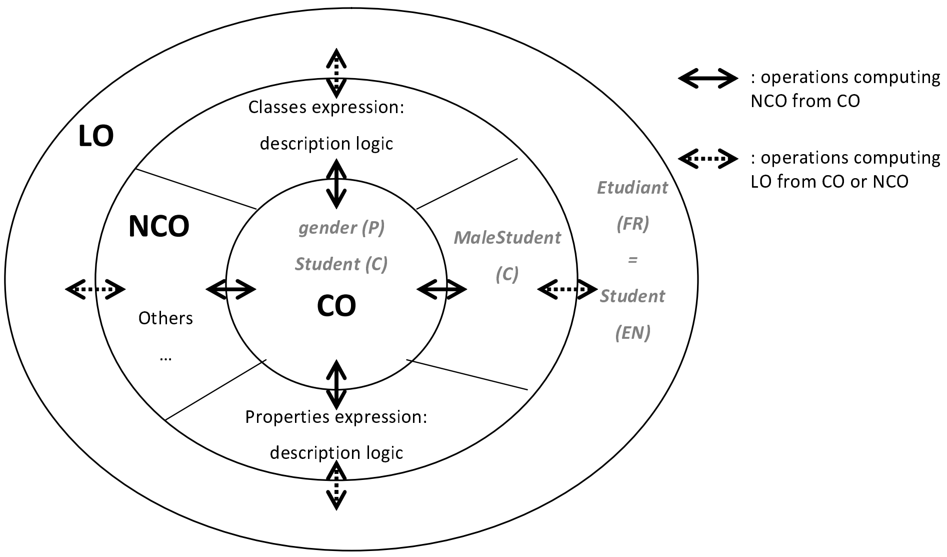


Figure 6.1: Taxonomy of ontologies

- *Non canonical ontologies (NCOs)*: besides of defining primitive concepts, a non canonical ontology defines derived (non canonical) concepts i.e., elements of the ontology which are expressed in terms of other ones (derived classes and properties). The example of Figure 6.1 defines at the NCO layer a non canonical class (`MaleStudent`) which is a restriction of the `Student` class on the property `gender` having the value `Male`.
- *Linguistic ontologies(LOs)*: establish textual terms for each concept of an ontology and relationships between these terms (e.g., hyponym, antonym, synonym). The LO layer of Figure 6.1 determines textual terms associated to the `Student` class in French and English.

Next section introduces the notion of OBDB.

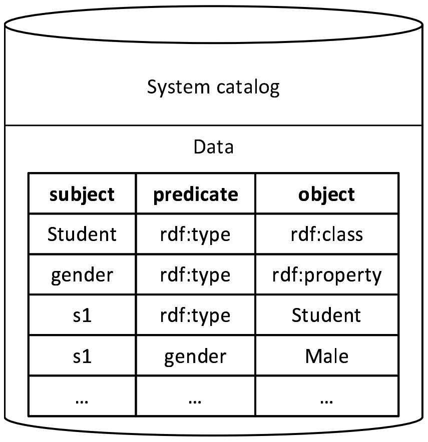
## 3 Ontology-based databases (OBDBs)

An OBDB is a database system that stores, in the same repository, ontologies together with the data they describe. An OBDB is equipped with an exploitation language that supports the creation and the manipulation (read, update and delete) of ontologies and their instances. As cited in Section 1 of this chapter, an OBDB is a specific PMMS dedicated to ontologies.

Different types of OBDBs have been proposed. We classify them into 3 types regarding their architecture and the capabilities provided for the support of non canonical concepts.

### 3.1 Type1 OBDBs

Type1 OBDBs (Figure 6.2) are used to store RDF data that may contain ontology descriptions together with their instances. Main RDF OBDBs are 3Store [Harris and Gibbins, 2003], Jena [Carroll et al., 2004], Oracle [Chong et al., 2005] or Sward [Petrini and Risch, 2007]. These OBDBs follow the simple

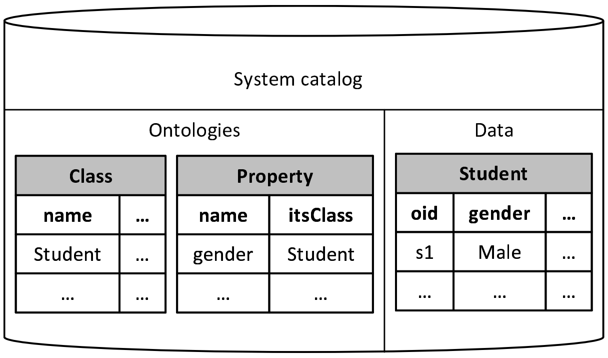


| subject | predicate | object |
|---|---|---|
| Student | rdf:type | rdf:class |
| gender | rdf:type | rdf:property |
| s1 | rdf:type | Student |
| s1 | gender | Male |
| ... | ... | ... |

Figure 6.2: Type1 OBDB

model of RDF to store data. Indeed, a single schema composed of a unique triple table `(subject, predicate, object)` is used to store both ontology descriptions and instance data. For ontology descriptions, the three columns of this table represent respectively subject ontology element identifier, predicate and object ontology element identifier. For example, the triple `(Student, rdf:type, rdf:class)` states that `Student` is an RDF class. Concerning instances data, which are stored in the same table, the three columns of this table represent respectively the instance identifier, the characteristic of an instance (i.e., property or class belonging) and the value of that characteristic. For example, the triple `(s1, gender, Male)` states that male is the gender of `s1` (an instance of the `Student` class). As RDF data may include RDFS or OWL ontology descriptions, most of these OBDBs provide a support for the semantics of RDFS or OWL. This semantics is usually hard-coded and thus non canonical concepts are supported using deductive rules [Chong et al., 2005] or external reasoners [Harris and Gibbins, 2003].

## 3.2 Type2 OBDBs



| Class | |
|---|---|
| name | ... |
| Student | ... |
| ... | ... |

| Property | |
|---|---|
| name | itsClass |
| gender | Student |
| ... | ... |

| Student | | |
|---|---|---|
| oid | gender | ... |
| s1 | Male | ... |
| ... | ... | ... |

Figure 6.3: Type2 OBDB

Type2 OBDBs (Figure 6.3) store separately ontology descriptions and instance data in two different schemes. Type2 OBDBs are PMMSs where the metamodel layer is hard-coded i.e., they support a fixed modeling formalisms. Main examples of type2 OBDBs are RDF Suite[Alexaki et al., 2001], Sesame [Broekstra et al., 2002], RStar [Lu et al., 2007], DLDB [Pan and Heflin, 2003] or OntoMS [Park et al., 2007]. The schema for ontology descriptions depends upon the ontology model used to represent ontologies (e.g., RDFS, OWL, PLIB). It is composed of tables used to store each ontology modeling primitive such as classes, properties and subsumption (inheritance) relationships. Concerning instance data, different schemes have been proposed that have different scalability characteristics. These OBDBs support mainly the usual subsumption semantics as specified in the RDFS semantics [Hayes, 2004] (i.e., `subClassOf` and `instanceOf` relationships).

To compute non canonical concepts, type2 OBDBs use different mechanisms like views [Pan and Heflin, 2003], labeling schemes [Park et al., 2007] or the subtable relationships issued from object-relational databases [Alexaki et al., 2001, Broekstra et al., 2002]. Some OBDBs address more complex reasoning using logic-based engines (e.g., Datalog engine) of deductive databases or OWL reasoners [Mei et al., 2006, Volz et al., 2005, Borgida and Brachman, 1993, Pan and Heflin, 2003].

## 3.3 Type3 OBDBs

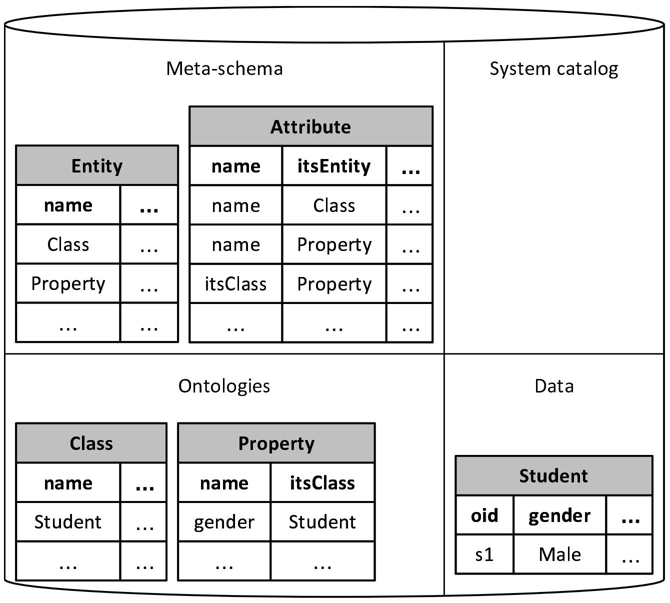


Figure 6.4: Type3 OBDB

Type3 OBDBs are PMMSs supporting an extensible metamodel layer so that we can define multiple modeling formalisms. The main example of Type3 OBDBs is OntoDB [Dehainsala et al., 2007]. This OBDB (Figure 6.4) proposes to add another schema to type2 OBDBs. This schema called *meta-schema* records metamodels of the supported ontology formalisms. For the ontology schema, the role played by the meta-schema is similar to the one played by the system catalog in traditional databases.

Before our extension of OntoDB, this OBDB used views and the associated database procedural language (PL/pgSQL) to compute non canonical concepts.

### 3.4 Synthesis

As we have seen in this section, studies on OBDBs have been mainly focused on the scalability of these new types of databases. Different types of representation of data have been proposed. Considering support of ontology, each OBDB supports the semantics of a given ontology model using hard-coded techniques either by using database mechanisms (e.g, views or relational-object operators) or by relying on an external logical engine. The aim of our work is to show the interest of our proposition by providing a more flexible approach that can be followed to support the semantics of several ontology models in different ways (e.g., with an internal procedure in the database system, with a call to an external reasoner or with the invocation of a web service).

As stated previously, OntoDB provides an interesting part, the meta-schema part, to support the evolution of the used ontology models. Thus, we suggest to exploit this schema for encoding the OWL formalism in OntoDB as it is presented in the next section.

## 4 Encoding the structural semantics of the OWL language

As we have previously outlined, we have chosen, as a running example, to show how OWL can be supported by the OntoDB/OntoQL system. In particular, we would like to show how this system can manage the behavior of non canonical concepts using our approach. This section shows how the OWL canonical (primitive) concepts of the OWL metamodel can be represented with OntoDB/OntoQL. For the sake of clarity, we take a simplified OWL metamodel which is presented in the figure 6.5.

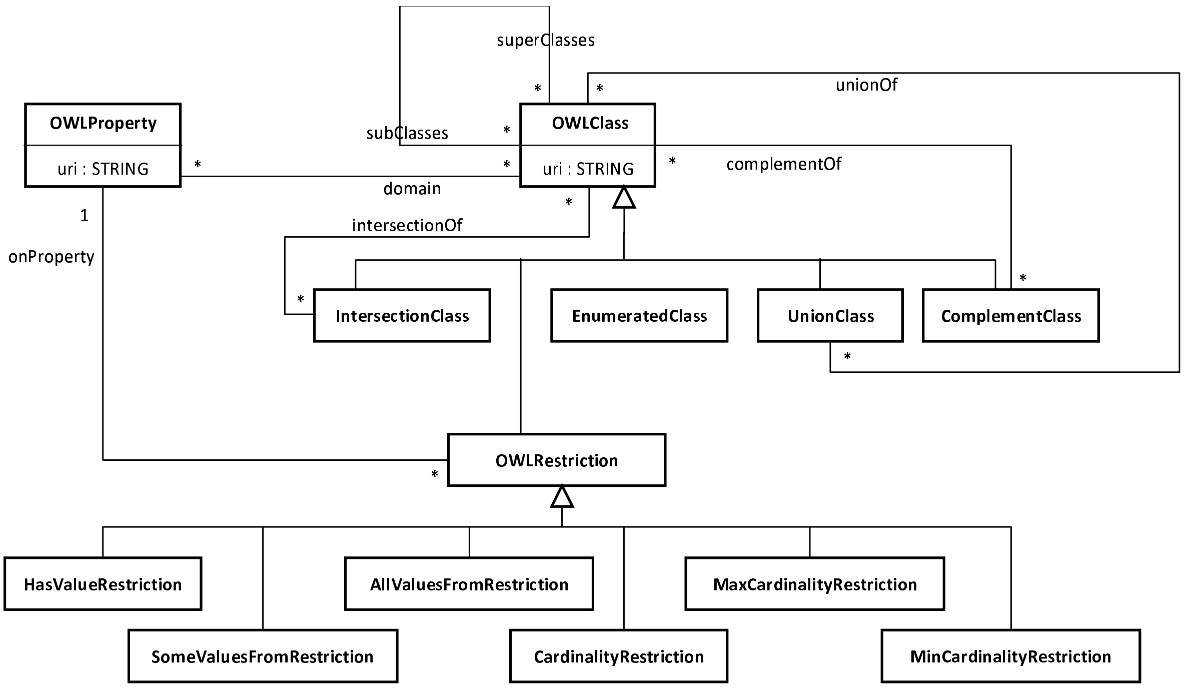


Figure 6.5: A simplified OWL metamodel

This metamodel contains two canonical concepts: `OWLClass` and `OWLProperty`. These concepts are primitive and cannot be derived from other concepts. Conversely, all the other concepts of this metamodel are non canonical. Indeed, they are derived concepts and have to be computed from other concepts. For instance, the result of an OWL union of classes is an OWL class made from the union of a set of OWL classes. Thus, `UnionClass` for instance is a non canonical concept. We use the proposition made in this thesis in order to compute these ontologies non canonical concepts.

The OntoQL language allows a user to create ontology models, ontologies and their instances, and to store the whole data related to these three levels in the corresponding parts of OntoDB. At this level, we are able to encode the statements that support the description of structural and descriptive semantics of the OWL metamodel. Listing 6.1 presents the statements for creating and storing the `OWLClass` and `OWLProperty` concepts.

Listing 6.1: Statements for creating the OWL metamodel

```
CREATE ENTITY #OWLClass (
 #uri STRING,
 #superClasses REF (#OWLClass) ARRAY,
 #subClasses REF (#OWLClass) ARRAY);

CREATE ENTITY #OWLProperty (
 #uri STRING,
 #domain REF (#OWLClass) ARRAY);
```

These statements create structures (tables) to store OWL classes and properties in OntoDB. Indeed, the `OWLClass` entity defines the concept of OWL class. It is described by an `uri`, a set of superclasses (`superClasses`) and a set of subclasses (`subClasses`). The `OWLProperty` defines the concept of OWL property that is characterized by an `uri` and has a `domain` (the classes which the property belongs to).

Now, let us consider the following example: a class `SchoolMember` could be defined from the union of `Professor` and `Student` classes. One possible semantics of the union of classes induces that the `SchoolMember` class becomes a super class of `Professor` and `Student` classes and conversely, `Professor` and `Student` become subclasses of `SchoolMember`. Besides, instances of the `SchoolMember` resulting class are obtained by computing the union of instances of `Professor` and `Student`. Here appears the need of an operator for computing the `SchoolMember` class. Next section exposes the support of non canonical concepts in OntoDB using our proposition.

## 5 Encoding operations for computing OWL non canonical concepts

Using the approach presented in Part II, we can define operations to compute the different OWL non canonical concepts. To illustrate our approach, we present the implementation of three chosen OWL non canonical constructors (`UnionClass`, `IntersectionClass` and `HasValueRestriction`) as the implementation of the other non canonical constructors is similar to the ones we present in this section.

### 5.1 Union of classes

`UnionClass` is an OWL class obtained from the union of a set of OWL classes. For instance, let us we consider that a class C1 is the union of two classes C2 and C3 *(C1 = C2 U C3)*. In the chosen semantics,

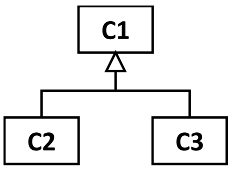


Figure 6.6: The OWL union class structure

the consequence of this union is that C1 becomes a super class of C2 and C3: *SubClasses(C1) = {C2, C3}* (Figure 6.6). Moreover, individuals (instances) of C1 are the union of the individuals of C2 and C3 *(Individuals(C1) = Individuals(C2) U Individuals (C3))*. Computing the OWL union of classes can be achieved using a semantic reasoner (e.g., Jena [Carroll et al., 2004]) or a program.

Listing 6.2 presents the OntoQL statement which creates the `UnionClass` concept.

Listing 6.2: Statement for creating the structure of the `UnionClass` structure

```
CREATE ENTITY #UnionClass
UNDER #OWLClass
 (#unionOf REF (#OWLClass) ARRAY);
```

This OntoQL statement defines the structure of this concept (i.e., an `UnionClass` has the same attributes of `OWLClass` and is defined by a set of classes).

In order to show the complete process of defining an union of classes, we create 2 OWL classes (`Professor` and `Student`). Listing 6.3 presents OntoQL statements creating these 2 classes.

Listing 6.3: Statements for defining OWL classes

```
CREATE #OWLClass Professor
PROPERTIES (
uri = 'http://www.lisi.ensma.fr/owlontology1#professor');

CREATE #OWLClass Student
PROPERTIES (
uri = 'http://www.lisi.ensma.fr/owlontology1#student');
```

Before processing the union of the `Professor` and `Student` classes, we need to define an operation for that purpose (Listing 6.4).

Listing 6.4: Statement for creating the `unionOf` operator

```
CREATE OPERATION #unionOf
INPUT (REF (#OWLClass) ARRAY)
OUTPUT (REF (#OWLClass));

CREATE IMPLEMENTATION #unionOfImp
DESCRIPTORS (
type = 'Java',
class = 'fr.ensma.lisi.owlncconcepts',
method = 'owlUnionOfClasses',
path = 'D:\\owlncoperators.jar')
IMPLEMENTS #unionOf;
```

The first statement defines an operation `unionOf` that takes an array of `OWLClass` as input and returns an `OWLClass`. Furthermore, we have set up an implementation (`unionOfImp`) of the defined operation. This implementation gives the metadata of a Java program allowing to process the union of classes. These metadata are necessary for the remote invocation of the Java program. The pseudo code of this program is given in Listing 6.5.

Listing 6.5: Extract of the pseudocode of the OWL union of classes operator

```
OWLClass owlUnionOfClasses(OWLClass[] C) {
    OWLClass cl = new UnionClass(C);
    return cl;
}

UnionClass(OWLClass[] C) {
    unionOf = C;
    updateOntologyStructure(C);
    computeUnionIndividuals(C);
}

void updateOntologyStructure() {

    //update ontology structure: classes of C as superClasses of cl
    for Ci in C {
        executeQuery("SELECT #superClasses FROM #OWLClass WHERE #name='" + cl.getName() + "'");
        resultSet.next();
        String superClasses = resultSet.getString(1).replace("{", "").replace("}", "");
        executeUpdate("UPDATE #OWLClass SET #superClasses=ARRAY[" +
                   superClasses.concat(", ").concat(cl.getOid()) +
                   "] WHERE #name='" + Ci.getName() + "'");
    }
}

void computeUnionIndividuals(OWLClass[] C) {
    for Ci in C {
        cl.addIndividuals(Ci.getIndividuals());
    }
}
```

*Meaning.* This program computes the OWL union of classes. Indeed, the `updateOntologyStructure` procedure organizes the classes in a hierarchy respecting the defined semantics of the union of classes. Besides, the `computeUnionIndividuals` procedure processes instances of the resulting class from the union operation.

Once the `unionOf` operation is defined and at least one associated implementation is set up, we can invoke this operation in OntoQL statements. Thus, the OntoQL statement for creating the `SchoolMember` class from the union of `Student` and `Professor` is presented in Listing 6.6.

Listing 6.6: Statement for creating a non canonical concept using a defined operator

```
CREATE #UnionClass SchoolMember
PROPERTIES (
uri = 'http://www.lisi.ensma.fr/owlontology1#schoolmember')
AS #unionOf (Professor, Student);
```

Notice that the way the operation is implemented is completely hidden to the OntoQL user.

The consequence of the execution of the previous statement is the modification of the structure of the ontology. Indeed, `SchoolMember` becomes a super class of `Professor` and `Student`, and conversely

Professor and Student classes become subclasses of SchoolMember. Moreover, the OWL *unionOf* operator defines SchoolMember individuals as the union of Professor and Student individuals.

Next subsection exposes the definition of the intersection of classes operator.

## 5.2 Intersection of classes

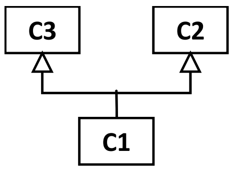


Figure 6.7: The OWL intersection class structure

IntersectionClass represents an OWL class that is obtained from the intersection of a set of OWL classes. The resulting class becomes a subclass of all the classes taking part of the intersection. Figure 6.7 illustrates the IntersectionClass concept with an example of a class C1 that is an intersection of C2 and C3 classes. Notice that in the chosen semantics, C1 is a subclass of C2 and C3 (*SuperClasses(C1) = {C2, C3}*). Therefore, C1 inherits the C2 and C3 properties.

Listing 6.7 presents the OntoQL statement which creates the IntersectionClass concept.

Listing 6.7: OntoQL statement for defining the structure of the IntersectionClass concept

```
CREATE ENTITY #IntersectionClass
UNDER #OWLClass
 (#intersectionOf REF (#OWLClass) ARRAY);
```

To be able to achieve intersection of classes, we need to define an operation for that purpose (Listing 6.8).

Listing 6.8: Statement for creating the unionOf operator

```
CREATE OPERATION #intersectionOf
INPUT (REF (#OWLClass) ARRAY)
OUTPUT (REF (#OWLClass));

CREATE IMPLEMENTATION #intersectionOfImp
DESCRIPTORS (
type = 'Java',
class = 'fr.ensma.lisi.owlncconcepts',
method = 'owlIntersectionOfClasses',
path = 'D:\\owlncoperators.jar')
IMPLEMENTS #intersectionOf;
```

The first statement defines an operation intersectionOf that takes an array of OWLClass as input and returns an OWLClass. The second statement defines an implementation (intersectionOfImp) of the defined operation. Listing 6.9 presents a part of the pseudo-code of the intersection operator.

Listing 6.9: Extract of the pseudocode of the OWL intersection of classes operator

```
OWLClass owlIntersectionOfClasses(OWLClass[] C) {
    OWLClass c1 = new IntersectionClass(C);
    return cl;
}

IntersectionClass(OWLClass[] C) {
    intersectionOf = C;
    updateOntologyStructure(C);
    computeIntersctionIndividuals(C);
}

void updateOntologyStructure() {

    //update ontology structure: classes of C as subClasses of cl
    for Ci in C {
        executeQuery("SELECT #subClasses FROM #OWLClass WHERE #name='" + cl.getName() + "'");
        resultSet.next();
        String subClasses = resultSet.getString(1).replace("{", "").replace("}", "");
        executeUpdate("UPDATE #OWLClass SET #subClasses=ARRAY[" +
                    subClasses.concat(", ").concat(cl.getOid()) +
                    "] WHERE #name='" + Ci.getName() + "'");
    }
}
```

If we consider that the class `StudentEmployee` is an intersection class of `Student` and `Employee` classes, the OntoQL statement executing the intersection is given in Listing 6.10.

Listing 6.10: Statement for computing an intersection class

```
CREATE #IntersectionClass StudentEmployee
PROPERTIES (
uri = 'http://www.lisi.ensma.fr/owlontology1#studentemployee')
AS #intersectionOf (Student, Employee);
```

The execution of the previous statement modifies the structure of the ontology. Indeed, `Student-Employee` becomes a subclass of `Employee` and `Student`, and conversely `Employee` and `Student` classes become super classes of `StudentEmployee`. Individuals of `StudentEmployee` are instances that are at the same time individuals of `Employee` and `Student` classes.

Next subsection exposes the definition of the has value restriction operator.

## 5.3 Has value restriction

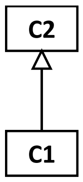


Figure 6.8: The OWL `HasValueRestriction` class structure

`HasValueRestriction` corresponds to an OWL class obtained by restricting an OWL class to a value of one of its properties. In the chosen semantics, the resulting class becomes a subclass of the class for which we apply the restriction. For instance, if C1 is a restriction of C2 on a property *p*, then C1 becomes a subclass of C2: *SuperClass(C1) = C2* (Figure 6.8), and instances of C1 are those of C2 having a fixed value for the property on which the restriction is applied.

Listing 6.11 presents the OntoQL statement which creates the `HasValueRestriction` concept.

Listing 6.11: Statement for defining the structure of the `HasValueRestriction` concept

```
CREATE ENTITY #HasValueRestriction
UNDER #OWLClass
 (#onProperty REF (#OWLProperty));
```

To be able to process a has value restriction, we need to define an operation for that purpose (Listing 6.12).

Listing 6.12: Statement for creating the `unionOf` operator

```
CREATE OPERATION #hasValueRestrictionOp
INPUT (REF (#OWLClass),
        REF (#OWLProperty),
        STRING value)
OUTPUT (REF (#OWLClass));

CREATE IMPLEMENTATION #hasValueRestrictionOpImp
DESCRIPTORS (
type = 'Java',
class = 'fr.ensma.lisi.owlncconcepts',
method = 'owlhasValueRestriction ',
path = 'D:\\owlncoperators.jar')
IMPLEMENTS #hasValueRestrictionOp;
```

The pseudo-code of the has value restriction operation is given in Listing 6.13.

Listing 6.13: Extract of the pseudocode of the OWL has value restriction operator

```
OWLClass owlHasValueRestriction(OWLClass C, OWLProperty P, String V) {
    OWLClass cl = new HasValueRestriction(C, P, V);
    return cl;
}

HasValueRestriction(OWLClass C, OWLProperty P, String V) {
    class = C;
    property = P;
    value = V;
    updateOntologyStructure(C);
    computeIntersctionIndividuals(C, P, V);
}

void updateOntologyStructure() {

    //update ontology structure: C as superClasses of cl

    executeQuery("SELECT #superClasses FROM #OWLClass WHERE #name='" + cl.getName() + "'");
    resultSet.next();
    String superClasses = resultSet.getString(1).replace("{", "").replace("}", "");
    executeUpdate("UPDATE #OWLClass SET #superClasses=ARRAY[" +
              superClasses.concat(", ").concat(cl.getOid()) +
              "] WHERE #name='" + Ci.getName() + "'");
}

void computeHasValueRestrictionIndividuals(OWLClass C, OWLProperty P, String V) {

        for indiv in (C.getIndividuals()) {
            if (indiv.P == value) {
                cl.addIndividuals(indiv);
            }
        }
    }
}
```

Let us consider that the `MaleStudent` class as a has value restriction of `Student` on the property `gender` having the value 'Male'. The OntoQL statement which creates the `MaleStudent` class is given in Listing 6.14.

Listing 6.14: Statement for computing a has value restriction class

```
CREATE #HasValueRestriction MaleStudent
PROPERTIES (
uri = 'http://www.lisi.ensma.fr/owlontology1#malestudent')
AS #hasValueRestrictionOp (Student, gender, 'Male');
```

The result of the execution of the previous statement modifies the structure of the ontology such that `MaleStudent` becomes a subclass of `Student`, and conversely `Student` becomes a super class of `MaleStudent`. Furthermore, individuals of `MaleStudent` are instances of `Student` having 'Male' value for the `gender` property.

## 6 Conclusion

In this chapter, we have presented a use case to show the interest of our approach which consists in extending PMMSs with behavioral semantics. This case study concerns the support of ontologies non canonical (derived) concepts in OBDBs which are specific PMMSs devoted to store and manipulate ontologies together with their instance data. In this work, we have shown that our proposition introduces dynamically operations that compute the derived concepts in the OBDB. These operations can be implemented differently e.g., with external reasoners, external programs, web services. This possibility can be useful as these implementations are adapted to specific settings (e.g., reasoners for small ontologies, or web services for distributed settings). The work presented in this chapter has been validated in [Bazhar et al., 2012a].

In the next chapter we go further this use case by showing how the approach of our thesis and the work accomplished in this chapter can be used to improve the design and the building of OBDBs.

Chapter 7

# Enhancing a method to design ontology-based databases

**Contents**

**Abstract.** As shown in the previous chapter, OBDBs can benefit from our approach to support non canonical concepts. These concepts are particularly important when designing OBDBs. Indeed, they have to be carefully managed in order to obtain a database in 3NF. Currently, most OBDB design methodologies process non canonical concepts with ad hoc programs which do not take into account the modifications that can be made in the ontology once the OBDB has been designed. Thus, in this chapter, we go further in our use case on OBDBs by showing how our proposition and the work accomplished in the previous chapter can be used to enhance an OBDB design methodology. This application has been validated in [Bazhar et al., 2012b].

## 1 Introduction

In the previous chapter, we have presented a use case of our proposition for the support of non canonical (derived) concepts in OBDBs. Indeed, we have shown that operations of PMMSs can be used to compute these non canonical concepts in OBDBs. In this chapter, we go further in this use case. We focus on the design of OBDBs, and particularly handling non canonical concepts in the OBDB design process.

Since an OBDB is a database, it shall be built following the classical steps of building traditional databases i.e., the conceptual, then the logical and finally the physical design. However, two main differences exist between the design of a traditional database and the design of an OBDB. Indeed, (1) a specific part of or the complete ontology plays the role of the conceptual model for designing the OBDB, and (2) both data and the ontology that describes the semantics of these data are stored in the OBDB.

Knowing that ontologies may be composed of canonical (primitive) and non canonical concepts, these non canonical concepts need to be identified. Indeed, some existing OBDB design approaches (e.g., [Chakroun et al., 2011]) use generally reasoners to compute the derived classes and properties before building the OBDB. But, when the original ontology is updated (e.g., a new derived class added), the existing OBDBs do not provide any mechanism to update the new structure of the ontology. Consequently, the OBDB design process has to be completely replayed taking into account the ontology updates.

Even if updates occur on an ontology, the OBDB shall offer flexible mechanisms to update the structure of the ontology while keeping the database in 3NF. In this chapter, we show that the support of operations in OBDBs (as OBDBs are specific PMMSs) can be useful to offer a flexible solution to compute non canonical concepts both in the design process and also if updates occur on ontologies. The work achieved in this chapter has been validated in [Bazhar et al., 2012b].

The remainder of this chapter is organized as follows. Section 2 presents the OBDB design methodology considered to support non canonical concepts using ad hoc mechanisms. Then, Section 3 shows an application of our approach to improve this OBDB design methodology. Finally, Section 4 concludes this chapter.

## 2 The considered OBDB design methodology

Since an OBDB is a database, it should be designed according to the classical design process dedicated to the development of databases identified in the ANSI/X3/SPARC architecture [ans, 1975]. However, when exploring the database literature, most of the research efforts were concentrated on the physical design phase, where various storage models for ontological data were given. Rdfsuite [Alexaki et al., 2001], Jena [Carroll et al., 2004], OntoDB [Dehainsala et al., 2007], Sesame [Broekstra et al., 2002], Owlgres [Stocker and Smith, 2008], SOR [Lu et al., 2007], Oracle [Das et al., 2004], etc. are examples of these systems. As OBDBs are more and more used, the proposition of a concrete design methodology, as in traditional databases, becomes a crucial issue. This development needs to follow the main steps of traditional database design approaches: *conceptual*, *logic* and *physical* designs. Chakroun and al. [Chakroun et al., 2011] proposed a five steps methodology for designing OBDBs. It starts from a conceptual model to provide logic and physical models following a five steps method. In this work, two types of ontological classes are identified: (1) canonical (primitive) and (2) non canonical (derived) classes. For the first type of classes, Chakroun and al. proposed a complete mechanism to manage canonical concepts structure and instances. Conversly, for non canonical concepts, Chakroun and al. proposed only the placement of classes in a subsumption hierarchy and the representation of non canonical instances using views. This placement is done after an inference step on the ontology achieved by a semantic reasoner. This proposition misses an OBDB-integrated mechanism for representing non canonical classes structure and instances views. Thus, we propose in this chapter, to extend OBDBs design methodology with a generic support of non canonical concepts based on our proposition that could be used to complete this methodology.

In this section, we introduce essential concepts to facilitate the understanding of the enhanced OBDB design methodology. Then, we describe the proposed methodology with its limitations. Finally, we show the interest of the OBDB extension we have achieved through a scenario.

## 2.1 Ontology dependencies

Functional dependencies play a key role in traditional database design. Since ontologies include classes and properties, two categories of dependencies are identified: (1) Class dependencies and (2) Property dependencies [Chakroun et al., 2011].

### 2.1.1 Class dependencies

Two types of class dependencies are distinguished.

- *Instance Driven Class Dependencies (IDCDs)*: a functional dependency among two concepts $C_1$ and $C_2$ ($C_1 \rightarrow C_2$) exists if each instance of $C_1$ determines one and only one instance of $C_2$. [Romero et al., 2009] proposed an algorithm to discover IDCDs among concepts of an ontology by exploiting the inference capabilities of DL-Lite. For instance, if we consider a functional role $mastersDegreeFrom$ with the $Person$ and $University$ classes as domain and range respectively, the functional dependency (FD) $Person \rightarrow University$ is defined. It means that the knowledge of a person with a valued property *masterDegreeFrom* determines a knowledge of one instance of $University$ class.

- *Static Dependencies (SDs)*: In SD, FD are defined between classes based on their definitions. A SD between two concepts $C_i$ and $C_j$ ($C_i \longmapsto C_j$) exists if $C_j$ can be derived from $C_i$. This definition can be based on a set of OWL [Dean and Schreiber, 2004] constructors (e.g., `owl:unionOf`, `owl:intersectionOf`, `owl:hasValue`). For example, if we consider a *level* property having as domain the $Student$ class, a class *MasterStudent* may be defined as a restriction on the $Student$ class having the value *master* for the *level* property. Therefore, the dependency $Student \longmapsto MasterStudent$ is obtained. It means that the knowledge of the whole instances of the $Student$ class determines the knowledge of the whole instances of the $MasterStudent$ class.

### 2.1.2 Property dependencies

As in traditional databases, functional dependencies between properties have been identified in the ontology context [Bellatreche et al., 2011, Calbimonte et al., 2009]. In [Calbimonte et al., 2009], authors proposed a formal framework for handling FD constructors for any type of OWL ontology. In [Bellatreche et al., 2011], the existence of FDs involving simple properties of each ontology class is assumed. For instance, if we consider the properties $idProf$ and *name* having as domain the $Professor$ class and describing respectively the professor identifier and name, the FD $idProf \rightarrow name$ may be defined. It means that the knowledge of the value of the professor identifier $idProf$ determines the knowledge of a single value of *name*.

## 2.2 Description of the design methodology

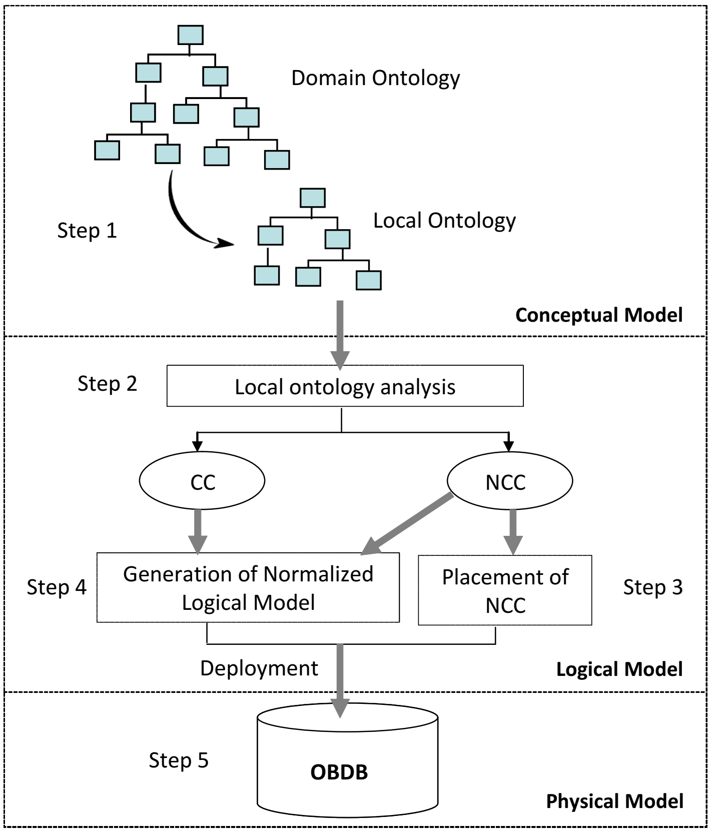


Figure 7.1: OBDB design approach

In [Chakroun et al., 2011], Chakroun and al. proposed a methodology which enriches the traditional database design process. It starts from a conceptual model to provide logical and physical models following a five steps method as described in Figure 7.1.

- **step1:** the designer extracts a fragment of a domain ontology $O$ (assumed available) (called Local Ontology(*LO*)) according to his/her requirements. The LO plays the role of the *conceptual model* (CM).

```
input  : $O$: a domain ontology ;
output: $normalized\ OBDB$
Extract the local ontology LO;
Analyse the LO and identify $CC^{LO}$ and $NCC^{LO}$;
Place the $NCC^{LO}$ in the appropriate subsumption hierarchy using an external reasoner;
foreach $CC_i^{LO} \in CC^{LO}$ do
    Generate the normalized tables (3NF);
    Generate a relational view defined on these tables;
end
foreach $NCC_i^{LO} \in NCC^{LO}$ do
    Generate a class view (a DL expression) on its related canonical class(es) ;
end
Choose any existing database architecture;
Deploy ontological data;
```

**Algorithm 1:** OBDB design algorithm

- **step2:** the resulting local ontology is then analyzed automatically to identify canonical ($CC^{LO}$) and non canonical classes ($NCC^{LO}$) by exploiting class dependencies. Based on the obtained $CC^{LO}$ and $NCC^{LO}$, two further steps are defined in parallel (steps 3 and 4).
- **step3:** concerns the placement of the $NCC^{LO}$ in the OBDB taking into account the subsumption hierarchy of classes. The complete subsumption relationship for the ontology classes is produced by a reasoner such as Racer[5], Pellet [Sirin and Parsia, 2004], etc.
- **step4:** addresses the generation of the normalized logical model for each ontological class where: (i) a set of normalized tables (3NF) are generated for each $CC_i$; (ii) a relational view is associated to each $CC_i \in CC^{LO}$ and (iii) a class view (a DL expression) on the canonical class(es) is associated to each $NCC_i \in NCC^{LO}$.
- **step5:** once the normalized logical model is obtained and $NCC^{LO}$ are placed in the subsumption relationship, the database administrator may choose an existing database architecture offering the storage of ontology and ontological data (Step5) for making persistent data describing metamodel, model and instances.

Algorithm 1 summarizes these steps. The following observations on this approach can be made:

- the designer needs to be familiar with the use of reasoners in order to establish the complete subsumption relationships between ontological classes;
- once the class hierarchy is persisted in the target database, no updates to the original ontology can be made;
- no detail has been given for class views computation.

Thus the defined approach has been hard-coded in the OBDB management system. In next section, we show how our proposition to define behavioral semantics of models elements can be used to enhance this methodology.

[5]http://www.racer-systems.com/

## 3 Enhancing the OBDB design methodology

In this section, we show how the proposed methodology to design OBDBs taking into account (1) the different phases of classical design approach and (2) offering the definition of the behavioral semantics of model elements, can be handled by the approach defined in Chapter 6. We focus only on classes and properties dependencies and how these concepts can be enriched using operations. Note that the OntoDB/OntoQL PMMS is used as storage model architecture for the *physical design phase*.

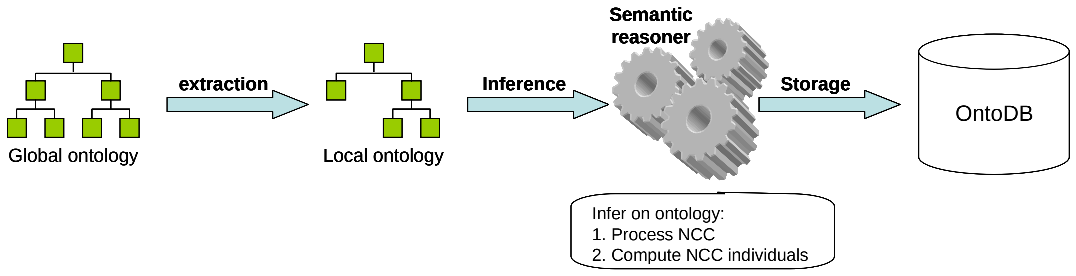


Figure 7.2: Our initial OBDB design methodology

Basically, the OBDB design methodology uses reasoners on ontologies to infer the non canonical concepts instances before building the logical model (Figure 7.2).

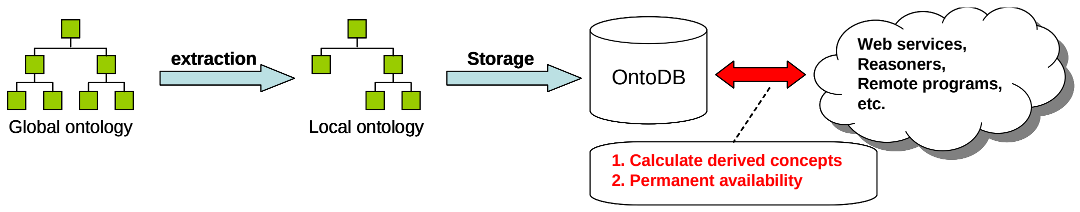


Figure 7.3: OBDB design methodology supporting non canonical concepts

By exploiting the support of non canonical concepts defined in Chapter 6 in the OBDB design methodology, we offer a dynamic and a flexible way to compute non canonical concepts (Figure 7.3). Indeed, if the ontology is updated, we can use the defined operators to infer on the ontology and/or calculate the new eventual derived concepts. This approach avoids to restart the OBDB design process in order to rebuild the logical model.

### 3.1 Explicit class dependencies with operations.

To show a real use case of the proposed methodology, the extension of the initial ontology model stored in the meta-schema part of OntoDB is required. Thus, we first enrich the meta-schema to handle (a) class dependencies, and (b) operations expressing behavioral semantics of non canonical concepts.

In order to handle class dependencies in the OntoDB/OntoQL PMMS, we extend the meta-schema part using statements of Listing 7.1.

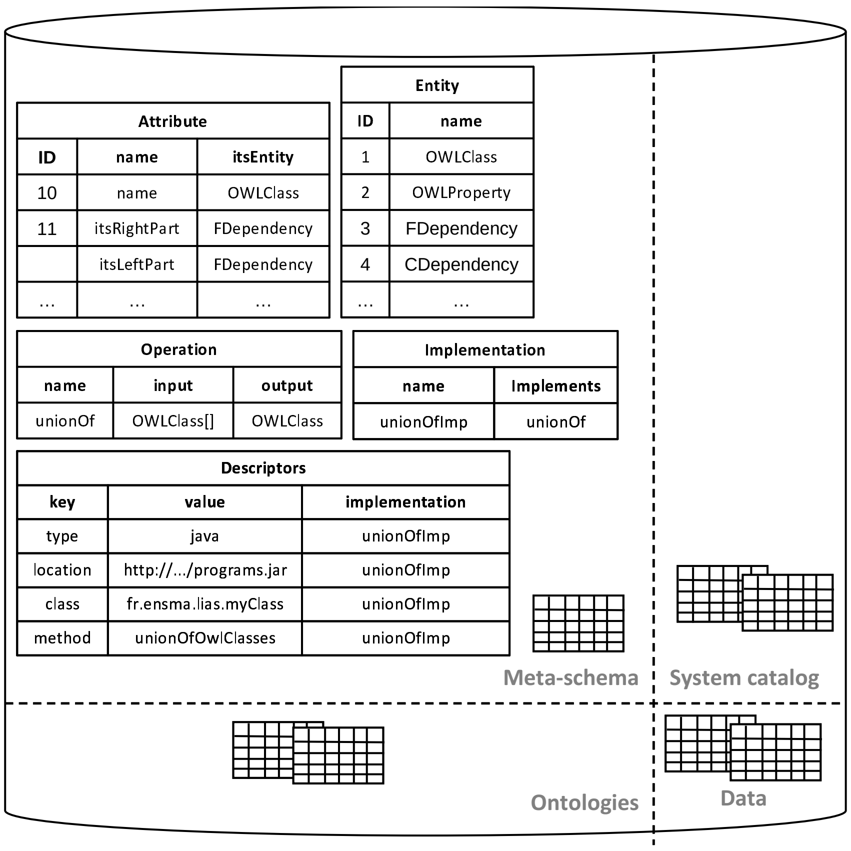
Listing 7.1: Statements for creating class dependency elements

```
CREATE ENTITY #CLeftPart (
                #itsClasses REF (#OWLClass) ARRAY);

CREATE ENTITY #CRightPart (
                #itsClass REF (#OWLClass));

CREATE ENTITY #CDependency (
                #rightPart REF (#CRightPart),
                #leftPart REF (#CLeftPart),
                #operator REF (#Operation));
```

These statements extend the meta-schema part of OntoDB with three entities in order to handle class dependencies including the left parts, the right part and the operation used to compute the right part class. Indeed, operations help us to define precisely the nature of the defined class dependencies and indicate the operators used to compute the resulting class of the dependency. As example, the statement of Listing 7.2 expresses and stores the class dependency $(Professor, Student \rightarrow SchoolMember)$. It states that `Professor` and `Student` classes determine together the `SchoolMember` class using the `unionOf` operator.

Listing 7.2: Statement for defining a class dependency

```
CREATE #CDependency (
        #leftPart (Professor, Student),
        #rightPart (SchoolMember),
        #operator (unionOf));
```



Attribute

| ID | name | itsEntity |
|---|---|---|
| 10 | name | OWLClass |
| 11 | itsRightPart | FDependency |
| | itsLeftPart | FDependency |
| ... | ... | ... |

Entity

| ID | name |
|---|---|
| 1 | OWLClass |
| 2 | OWLProperty |
| 3 | FDependency |
| 4 | CDependency |
| ... | ... |

Operation

| name | input | output |
|---|---|---|
| unionOf | OWLClass[] | OWLClass |

Implementation

| name | Implements |
|---|---|
| unionOfImp | unionOf |

Descriptors

| key | value | implementation |
|---|---|---|
| type | java | unionOfImp |
| location | http://.../programs.jar | unionOfImp |
| class | fr.ensma.lias.myClass | unionOfImp |
| method | unionOfOwlClasses | unionOfImp |

Figure 7.4: Extract of the meta-schema deployment

This dependency will be used to derive the non canonical `SchoolMember` class as the union of `Professor` and `Student` classes.

## 3.2 Explicit property dependencies with operations.

To support functional dependencies, the meta-schema part has to be extended with entities storing functional dependencies (Listing 7.3).

Listing 7.3: Statements for creating property dependency elements

```
CREATE ENTITY #FLeftPart (
            #itsProps REF (#OWLProperty) ARRAY);

CREATE ENTITY #FRightPart (
            #itsProp REF (#OWLProperty));

CREATE ENTITY #FDependency (
            #rightPart REF (#FRightPart),
            #leftPart REF (#FLeftPart),
            #operator REF (#Operation),
            #itsClass REF (#OWLClass));
```

Similarly to class dependencies, the statements above extend the meta-schema with three entities to handle functional dependencies between properties of a given class. A functional dependency is characterized by a left part which is defined by one or many properties, a right part property, an eventual operator to calculate the right part property if it is a derived property and the class on which the dependency is defined.

The statements of Listing 7.4 create two functional dependencies. The first one expresses the dependency between `idProf` and `name` properties of the `Professor` class, and the second one expresses the dependency between `birthday` and `age` properties of `Professor`. In the second dependency, the `age` property is computed using the `calculateAge` operator. This dependency will be used to derive a 3NF schema of tables associated to the `Professor` class.

Listing 7.4: Statements for defining a property dependency

```
CREATE #FDependency (
       #leftPart (idProf),
       #rightPart (name),
       #itsClass (Professor));

CREATE #FDependency (
       #leftPart (birthday),
       #rightPart (age),
       #operator (calculateAge),
       #itsClass (Professor));
```

Figure 7.4 shows the meta-schema deployment in which the ontology model and operations are stored, and Figure 7.5 shows the deployment of the ontology in which ontologies together with classes and property dependencies are stored.

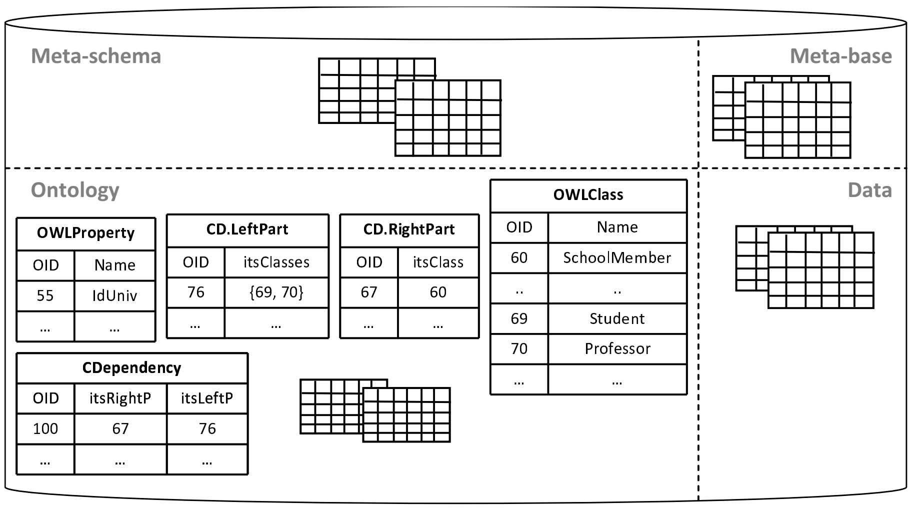


Figure 7.5: Extract of the ontology deployment

## 3.3 Generating the logical model of an OBDB.

Once operations are created and stored in the OBDB, and classes and property dependencies are expressed, we can generate a complete logical model of the OBDB including both a 3NF schema for each canonical class and a class view for each non canonical class.

To compute non canonical concepts, we invoke an existing operation. For example, the `SchoolMember` non canonical class is computed using the statement of Listing 7.5.

Listing 7.5: Statement for defining a non canonical concept

```
CREATE #OWLClass SchoolMember
AS unionOf (Professor, Student);
```

To generate the logical model of the OBDB, we invoke an operation which executes a program implementing the algorithm 1. The logical model of the OBDB under design could be obtained with the statement of Listing 7.6.

Listing 7.6: Statements for building the OBDB schema

```
CREATE #LogicalSchema
AS algorithm();
```

With regard to the `SchoolMember` class, a class view is computed based on its definition ($SchoolMember \equiv Professor \cup Student$). This view is associated to the non canonical concept. Figure 7.6 shows an example of the deployment of the generated normalized logical model of the `Professor`, `Student` and `SchoolMember` classes in the OntoDB/OntoQL PMMS.

## 4 Conclusion

In this chapter, we have shown that our proposition to manage the behaviors of models elements can be useful for designing OBDBs. Indeed, existing OBDB design methodologies do not handle the modifi-

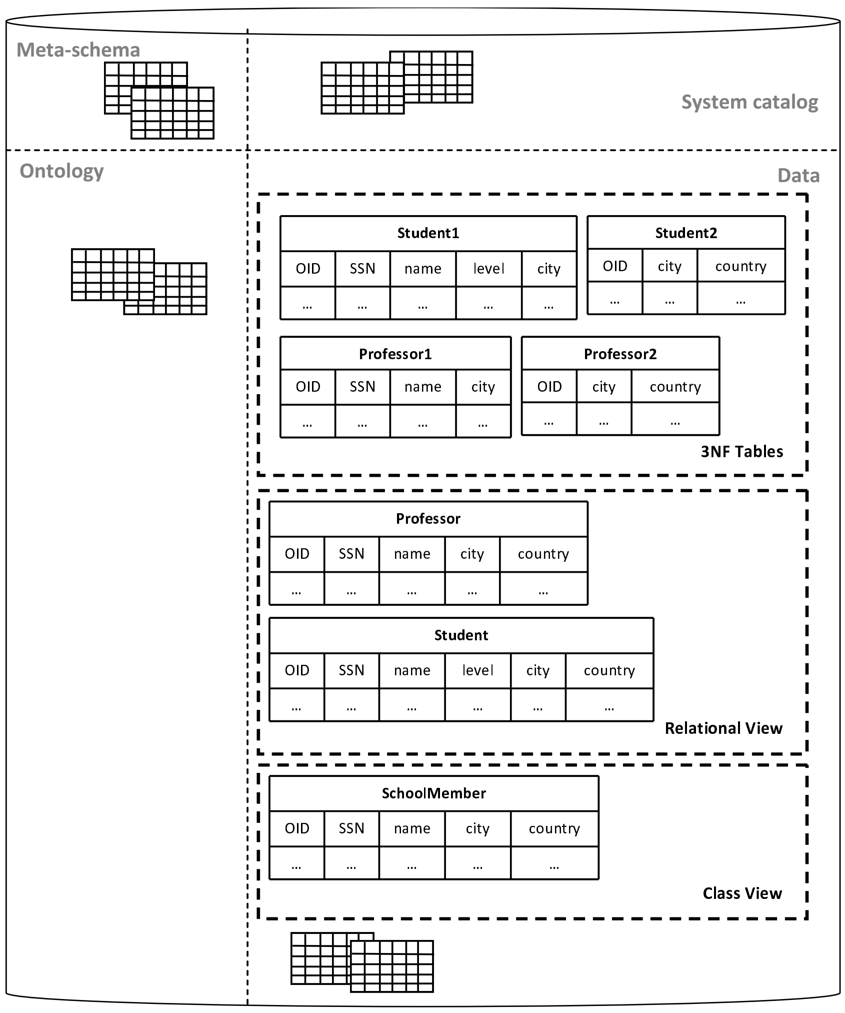


Figure 7.6: Extract of the data deployment

cation of the original ontology. By attaching operations to functional dependencies, this problem can be solved. Moreover, as these operations can be implemented in different manners, the enhanced design methodology is not restricted to the usage of reasoners which do not scale for large ontologies. The work presented in this chapter has been validated in [Bazhar et al., 2012b].

Having shown the interest of our proposition in the context of OBDB in the two last chapters, we present another case study in the next chapter which aims at transforming models for real-time applications.

Chapter 8

# Model transformation and model analysis

**Contents**

**Abstract.** In the two previous chapters, we have seen the interest of our approach (presented in Part II) for specific cases of PMMSs which are ontology-based databases. To show that our approach can be used in a wide range of contexts, we propose in this chapter a case study concerning a model transformation and model analysis applied to real-time model analysis. This use case has been validated in [Bazhar et al., 2013b].

## 1 Introduction

In the two previous chapters, we have shown the usefulness of our proposition in the context of ontology-based databases (OBDBs) which are specific PMMSs dedicated to store and manipulate ontologies. Indeed, we have seen that, thanks to our approach, OBDBs become able to compute non canonical (derived) concepts using flexible mechanisms. Then, we have shown that our proposition is useful to improve the OBDB design process by integrating the dynamic mechanisms offered by our approach. A last use case showing the utility of our proposition is presented in this chapter.

In the context of real-time and embedded systems, different languages and formalisms are dedicated to design models of real-time and embedded systems (e.g., MARTE [mar, 2011] and AADL [aad, 2012]). These formalisms provide different points of view of real-time systems. For instance, AADL is dedicated to design the architecture of embedded systems while MARTE is devoted to design the hardware and software aspects of real-time systems and to provide capabilities for real-time analysis (particularly the analysis of the schedulability of real-time systems). As a consequence, such formalisms propose various methodologies and provide different constructors to design real-time systems leading to the heterogeneity of models concerning the same system.

However, real applications use models for multiple purposes like data exchange, models and data sharing, analysis, etc., and these tasks have to be accomplished independently of the formalisms used to design models. For instance, the analysis of an AADL model must go through a model transformation to MARTE since AADL does not offer analysis capabilities. Achieving such tasks requires model mappings and/or transformations.

In this chapter, we show the usefulness of our proposition for model transformation and model analysis. Indeed, we show how our approach supports the transformation of AADL models to MARTE models in PMMSs, then we demonstrate how this approach can be used to analyze models of real-time systems from the PMMS. This work has been validated in [Bazhar et al., 2013b].

The remainder of this chapter is organized as follows. Section 2 introduces a motivating example that shows the difference of AADL and MARTE formalisms. Section 3 presents the persistence of AADL and MARTE in OntoDB for the support of both modeling formalisms. Section 4 introduces model transformation examples that show the usefulness of extending PMMSs with operations. Section 5 addresses the use of operations for model analysis. Finally, section 6 is devoted to a conclusion.

## 2 Motivating example

This section presents the example we use throughout this chapter. We use this example as a case study to show the need of handling behavioral semantics in PMMSs.

### 2.1 A real-time system example

The aim of this example is to design an uniprocessor system with three periodic tasks (*T1*, *T2* and *T3*). Each task is characterized by a period *P*, a deadline *D*, and a worst-case execution time *ET*. The system scheduling follows the EDF (Earliest Deadline First) scheduling policy. This system is defined as a set of tasks: $S = T1, T2, T3$, where:

$T1 =< P = 29ms, D = 29ms, ET = 7ms >$

$T2 =< P = 5ms, D = 5ms, ET = 1ms >$

$T3 =< P = 10ms, D = 10ms, ET = 2ms >$

This kind of systems can be designed using languages dedicated to design real-time and embedded systems like AADL [aad, 2012] or MARTE [mar, 2011]. We represent the system described above using both languages in order to show the difference of representing the same system using different languages. Before this, next sections expose briefly the AADL and MARTE languages.

## 2.2 The AADL language

AADL (Architecture Analysis and Design Language) is an architecture description language. The architecture description through AADL consists in describing components and their hierarchical composition. Indeed, three categories of components exist: software, hardware and system components. Moreover, AADL provides three equivalent modeling supports: the textual format, the graphical notation, and the XML format that eases processing AADL models with external tools.

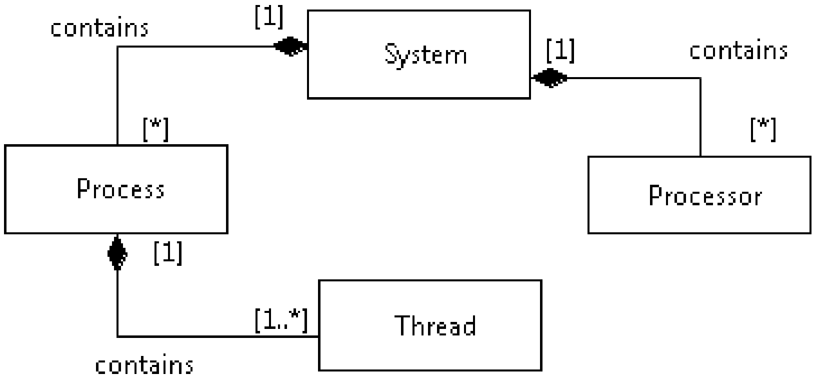


Figure 8.1: Part of the architecture of an application designed with AADL

In our example, we only use a subset of AADL elements. Figure 8.1 represents these AADL elements and their interactions. A system in AADL is defined a set of processes and a set of processors. Each process may contain multiple threads. Each element (in Figure 8.1) is itself represented by a classifier, an element type and its implementations. For instance, the `Thread` element is represented by `ThreadClassifier`, `ThreadType` and `ThreadImpl` (Figure 8.1), and only implementations can have subcomponents.

We use the textual format of AADL to design our system *S* using OSATE editor[6]. Figure 8.3 shows the representation of our system using the textual format of AADL. The system implementation is called `embSys.Impl` (system component) and contains two subcomponents. The first one is the `cpu` processor (hardware component) which is an instance of the `cpu_embSys.Impl` processor implementation. The second subcomponent is the `proc` process (software component) which is allocated to the `cpu` proces-

[6]http://www.aadl.info

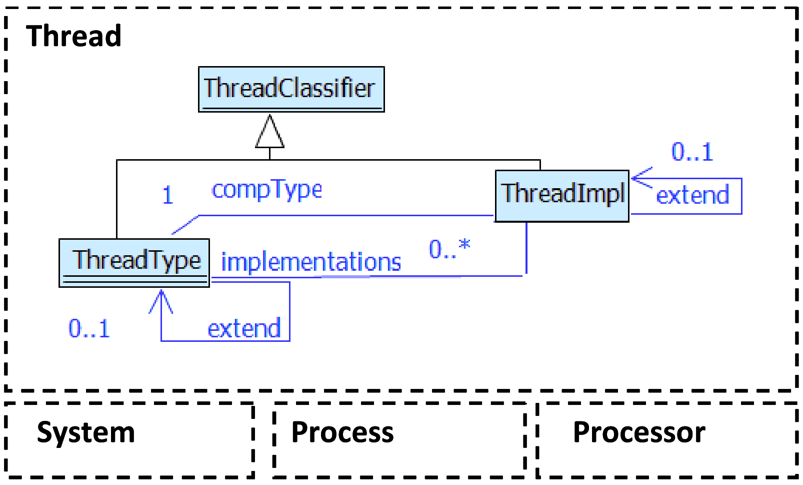


Figure 8.2: A subset of AADL metamodel for threads

sor for execution. `proc` is an instance of the `process_embSys.Impl` process implementation which contains three tasks: `T1`, `T2` and `T3` (software components), where each task is an instance of a thread implementation. All threads implementations of our example implement the same thread type. However, each thread implementation is characterized by different properties (period, deadline and execution time).

```
system embsys
end embsys;
system implementation embsys.Impl
  subcomponents
    cpu: processor cpu_embsys.Impl;
    proc: process process_embsys.Impl;
  properties
    Actual_Processor_Binding =>  reference cpu applies to proc;
end embsys.Impl;
processor cpu_embsys
end cpu_embsys;
processor implementation cpu_embsys.Impl
  properties
    Scheduling_Protocol => EDF;
end cpu_embsys.Impl;
process process_embsys
end process_embsys;
process implementation process_embsys.Impl
  subcomponents
    T1: thread Task.Impl1;
    T2: thread Task.Impl2;
    T3: thread Task.Impl3;
end process_embsys.Impl;
```

```
thread Task
end Task;
thread implementation Task.Impl1
  properties
    Dispatch_Protocol => Periodic;
    Compute_Execution_Time => 7 Ms .. 7 Ms;
    Deadline => 29 Ms;
    Period => 29 Ms;
end Task.Impl1;
thread implementation Task.Impl2
  properties
    Dispatch_Protocol => Periodic;
    Compute_Execution_Time => 1 Ms .. 1 Ms;
    Deadline => 5 Ms;
    Period => 5 Ms;
end Task.Impl2;
thread implementation Task.Impl3
  properties
    Dispatch_Protocol => Periodic;
    Compute_Execution_Time => 2 Ms .. 2 Ms;
    Deadline => 10 Ms;
    Period => 10 Ms;
end Task.Impl3;
```

Figure 8.3: The system $S = T1, T2, T3$ expressed in an AADL

## 2.3 The MARTE language

While AADL is dedicated to design system architectures, MARTE is structured around two main concerns: (i) to design the features of real-time and embedded systems and (ii) to annotate application models in order to support the analysis of the system properties.

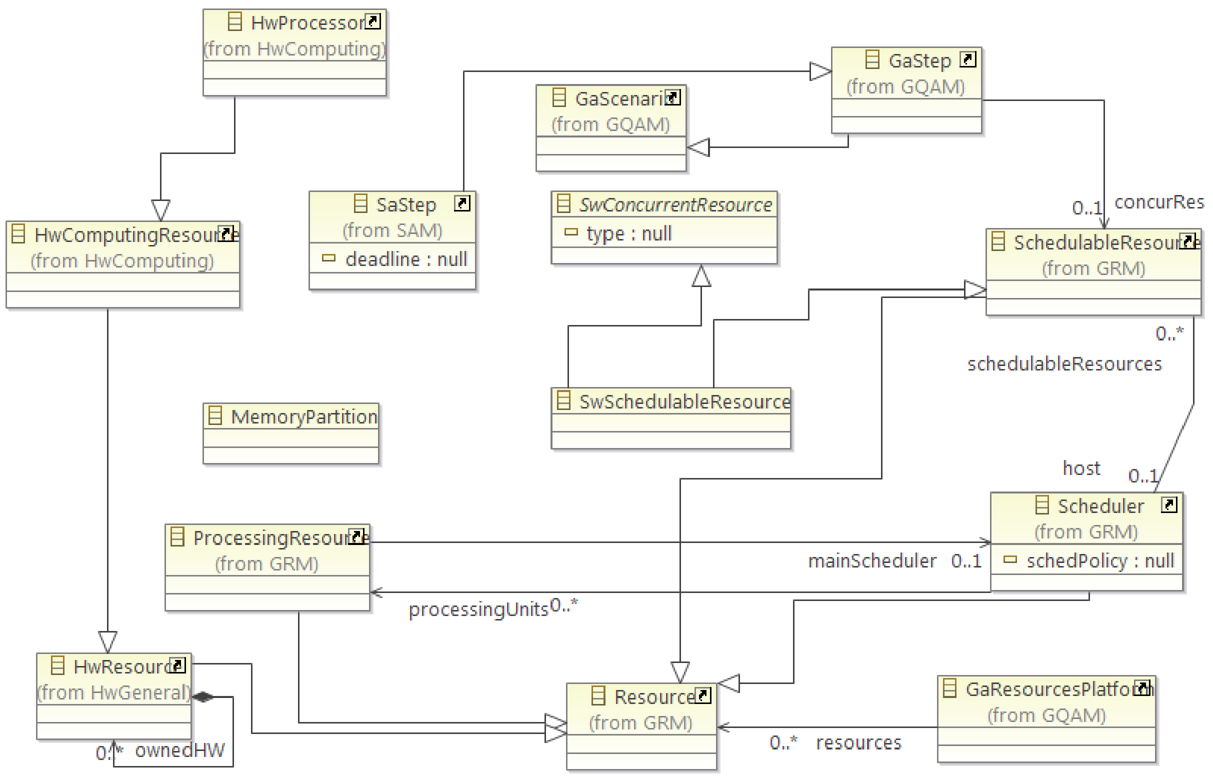


Figure 8.4: A subset of the MARTE metamodel

MARTE (Modeling and Analysis of Real Time and Embedded systems) is a modeling language dedicated to design both software and hardware aspects of real-time and embedded systems and enables schedulability analysis. The MARTE metamodel contains more than 130 classes [mar, 2011]. In the context of our work, we use only a subset of this metamodel that is shown in Figure 8.4 where `SwSchedulableResource` and `SaStep` express the elements of the software architecture of a real-time system. `MemoryPartition` and `Scheduler` express the operating system. This latter plays the role of a mapping layer between the software and the hardware architecture described by `HwProcessor`.

Since MARTE provides the facility to design models for real-time analysis, `SaAnalysisContext` and `GaResourcePlatform` concepts are dedicated to accomplish this task by instantiating elements of the software and hardware models according to a specific analysis context chosen by the designer.

The MARTE model corresponding to the system of our example is given in Figure 8.5.

## 2.4 AADL to MARTE transformation

As we have seen in the previous section, AADL and MARTE can express the same system using different constructors and following different methodologies. One of the major differences between these two languages is that AADL is more oriented towards architecture description and does not offer the capability to analyze the schedulability of systems, while MARTE meets this need. This example shows the problem of heterogeneous modeling that appears in the design of complex systems. Indeed, analyzing an AADL model schedulability must go through a model transformation to MARTE.

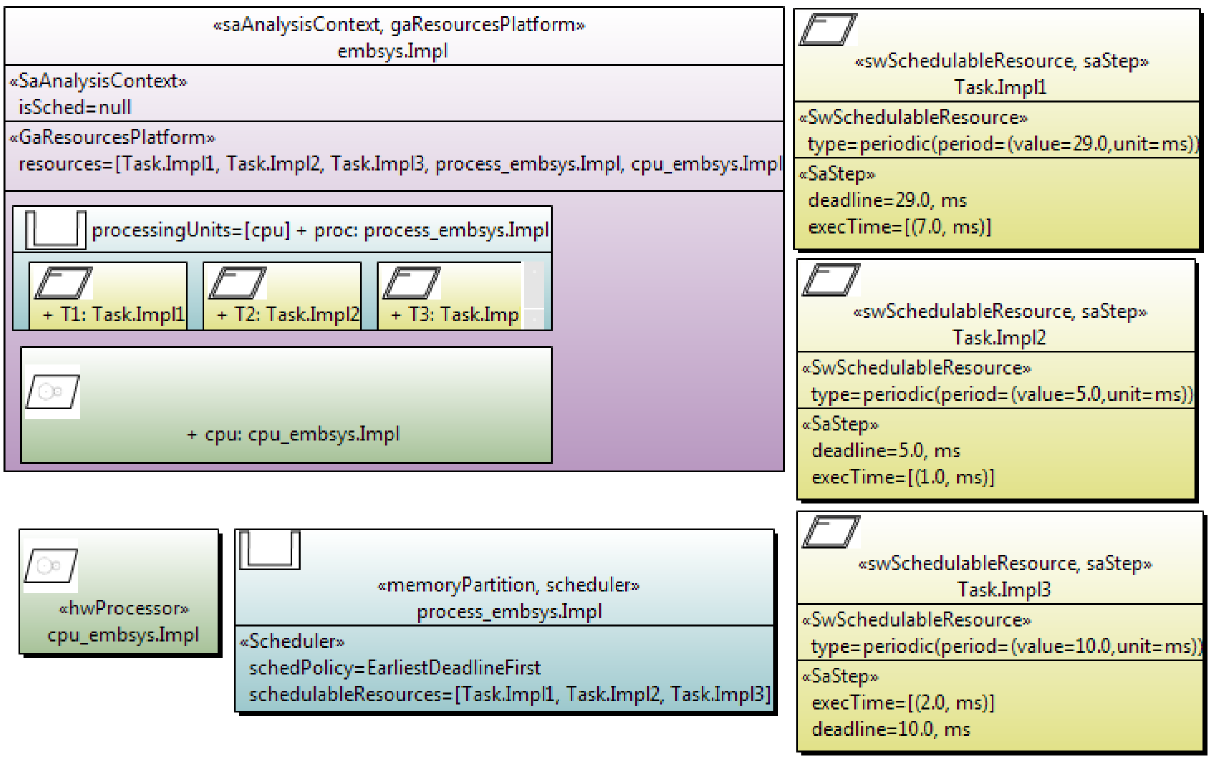

«saAnalysisContext, gaResourcesPlatform»
embsys.Impl
«SaAnalysisContext»
isSched=null
«GaResourcesPlatform»
resources=[Task.Impl1, Task.Impl2, Task.Impl3, process_embsys.Impl, cpu_embsys.Impl
processingUnits=[cpu] + proc: process_embsys.Impl
+ T1: Task.Impl1
+ T2: Task.Impl2
+ T3: Task.Imp
+ cpu: cpu_embsys.Impl
«swSchedulableResource, saStep»
Task.Impl1
«SwSchedulableResource»
type=periodic(period=(value=29.0,unit=ms))
«SaStep»
deadline=29.0, ms
execTime=[(7.0, ms)]
«swSchedulableResource, saStep»
Task.Impl2
«SwSchedulableResource»
type=periodic(period=(value=5.0,unit=ms))
«SaStep»
deadline=5.0, ms
execTime=[(1.0, ms)]
«hwProcessor»
cpu_embsys.Impl
«memoryPartition, scheduler»
process_embsys.Impl
«Scheduler»
schedPolicy=EarliestDeadlineFirst
schedulableResources=[Task.Impl1, Task.Impl2, Task.Impl3]
«swSchedulableResource, saStep»
Task.Impl3
«SwSchedulableResource»
type=periodic(period=(value=10.0,unit=ms))
«SaStep»
execTime=[(2.0, ms)]
deadline=10.0, ms

Figure 8.5: Our system expressed with MARTE

| | **AADL** | **MARTE** |
|---|---|---|
| rule 1 | `SystemType` and `SystemImpl` | `saAnalysisContext` and `gaResourcesPlatform` classes |
| rule 2 | `SystemClassifier` subcomponent | Specified by `Resources` property of `gaResourcesPlatform` class |
| rule 2.1 | `ProcessType` and `ProcessImpl` | `MemoryPartition` and `Scheduler` classes |
| rule 2.2 | `ProcessorType` and `ProcessorImpl` | `hwProcessor` class |
| rule 2.3 | `ProcessorImpl` properties<br><br>`scheduling_protocol` property | Specified by the `schedPolicy` property of `Scheduler` class. |
| rule 2.3.1 | `ProcessClassifier` subcomponent | `SchedulableResources` property of `Scheduler` class |
| rule 2.3.1.1 | `ThreadType` and `ThreadImpl` | `swSchedulableResource` and `saStep` classes |
| rule 2.3.1.2 | `ThreadImpl` properties:<br><br>- `dispatch_protocol` and `period`<br><br>- `deadline`<br><br>- `compute_execution_time` | Properties of `swSchedulableResource` class:<br>- Specified by `type` property which is a type of `ArrivalPattern`<br>In this case: properties of `saStep` class:<br>- Specified by `deadline` property<br>- Specified by `execTime` property |
| rule 2.4 | `SystemImpl` properties<br><br>- `actual_procesor_binding` | Specified by `processingUnits` property of `Scheduler` class |

Table 8.1: AADL to MARTE transformation rules

Table 8.1 summarizes the different rules for transforming AADL models to MARTE. These transformation rules are introduced in order to justify the operations, we define later in the chapter, for transforming AADL models to MARTE ones. For instance, in `rule1` an AADL system (the `SystemType` and `SystemImp` concepts) is transformed to `saAnalysisContext` and `gaResourcesPlatform` concepts in MARTE.

In the next section, we show how our approach can be used to handle the heterogeneity of real-time models (using model transformation operations) and process their schedulability in a PMMS.

## 3 Supporting AADL and MARTE in PMMS

### 3.1 Metamodels definition

The metamodel part of the OntoDB model repository can be enriched to support new metamodels using the OntoQL language. For each resource in the metamodel, an OntoQL statement is defined. Below we give the OntoQL statements for defining both AADL metamodel (Listing 8.1) and MARTE metamodel (Listing 8.2).

Listing 8.1: A subset of OntoQL statements for creating the AADL metamodel

```
CREATE CLASS #Property (
       #name STRING,
       #value STRING);

CREATE CLASS #SystemSubComponent;

CREATE CLASS #ProcessSubComponent;

// create the ThreadClassifier concept
CREATE CLASS #ThreadClassifier
UNDER #ProcessSubComponent;

// create the ThreadType concept
CREATE CLASS #ThreadType
UNDER #ThreadClassifier (
       #name STRING
       #extends REF (#ThreadType));

// create the ThreadImpl concept
CREATE CLASS #ThreadImpl
UNDER #ThreadClassifier (
       #name STRING,
       #properties REF (#Property) ARRAY,
       #implements REF (#ThreadClassifier),
       #extends REF (#ThreadImpl));

CREATE CLASS #ProcessClassifier
UNDER #SystemSubComponent;

CREATE CLASS #ProcessType
UNDER #ProcessClassifier (
       #name STRING);

CREATE CLASS #ProcessImpl
UNDER #ProcessClassifier (
        #name STRING,
        #subComponents REF (#ThreadClassifier) ARRAY,
        #implements REF (#ProcessType));

CREATE CLASS #ProcessorClassifier
UNDER #SystemSubComponent;

CREATE CLASS #ProcessorType
UNDER #ProcessorClassifier (
        #name STRING);

CREATE CLASS #ProcessorImpl
UNDER #ProcessorClassifier (
        #name STRING,
        #implements REF (#ProcessorType));

CREATE CLASS #SystemClassifier;

CREATE CLASS #SystemType
UNDER #SystemClassifier (
        #name STRING);

CREATE CLASS #SystemImpl
UNDER #SystemClassifier (
        #name STRING,
        #subComponents REF (#SystemSubComponent) ARRAY,
        #properties REF (#Property) ARRAY,
        #implements REF (#SystemType));
```

Listing 8.2: A subset of OntoQL statements for creating the MARTE metamodel

```
CREATE CLASS #SchedulableResource;

CREATE CLASS #swSchedulableResource
UNDER #SchedulableResouce, #swConcurrentResouce;

CREATE CLASS #HwProcessor
UNDER #HwComputingResouce;

CREATE CLASS #MemoryPartition;

CREATE CLASS #ProcessingResouce;

CREATE CLASS #HwResource;

CREATE CLASS #Resource;

CREATE CLASS #Scheduler (
         #schedulableResources REF (#swSchedulableResouce));
```

## 3.2 Model definition

Once a metamodel is defined and supported by the PMMS, it becomes possible to create models conforming to that metamodel. Statements of Listing 8.3 create the AADL model of our example.

Listing 8.3: A subset of OntoQL statements to create the AADL model

```
// create a ThreadType (Task)
```

```
CREATE #ThreadType Task;

// create a ThreadImpl (Task.Impl1)
CREATE #ThreadImpl Task.Impl1
        #PROPERTIES (
             Dispatch_Protocol = Periodic ,
             Compute_Execution_Time = 7Ms..7Ms,
             Deadline = 29Ms,
             Period = 29Ms)
        #IMPLEMENTS Task;

// create a ProcessType (process_embSys)
CREATE #ProcessType process_embSys;

// create a ProcessImpl (process_embSys.impl)
CREATE #ProcessImpl process_embSys.impl
        #SUBCOMPONENTS (
           T1 = Task.Impl1 ,
           T2 = Task.Impl2 ,
           T3 = Task.Impl3)
        #IMPLEMENTS process_embSys;

CREATE #ProcessorType cpu_embSys;

CREATE #ProcessorImpl cpu_embSys.Impl
        #PROPERTIES (Scheduling_Protocol = EDF)
        #IMPLEMENTS cpu_embSys;

CREATE #SystemType embSys;

CREATE #SystemImpl embSys.Impl (
        #SUBCOMPONENTS (
           cpu = cpu_embSys.Impl ,
           proc = process_embSys.Impl)
        #PROPERTIES (
              Actual_Processor_Binding = (cpu, proc))
        #IMPLEMENTS embSys;
```

Next section shows how our proposition can be used to transform AADL models to MARTE.

## 4 Transforming models within PMMS

This section shows how transformation operations of AADL models to MARTE ones can be defined. We precise that our objective is not to propose a new transformation approach, neither to guarantee a safe transformation from AADL to MARTE. Several work have addressed the transformation form AADL to MARTE. Our transformation is based on the work of [Mallet et al., 2009, mar, 2011].

Our objective is to use the possibility to introduce operations on the fly in the PMMS in order to transform AADL models into MARTE ones. To do so, we firstly create model transformation operations that will transform AADL concepts to MARTE ones, then we create corresponding implementations. These two steps are explained below in detail.

## 4.1 Definition of transformation operations

The definition of AADL to MARTE transformation operations consists in specifying for each operation its name, its eventual input and output types. The statements of Listing 8.4 define some of the essential transformation operations based on rules defined in Table 8.1.

Listing 8.4: AADL to MARTE transformation operations

```
CREATE OPERATION #rule1
INPUT (REF (#SystemType),          // AADL source elements
       REF (#SystemImpl))
OUTPUT (REF (#saAnalysisContext), // MARTE target elements
        REF (#gaResourcesPlatform));

CREATE OPERATION #rule2
INPUT (REF (#SystemSubComponent) ARRAY)
OUTPUT (REF (#Resource) ARRAY);

CREATE OPERATION #rule2.1
INPUT (REF (#ProcessType),
       REF (#ProcessImpl))
OUTPUT (REF (#MemoryPartition),
        REF (#Scheduler));

CREATE OPERATION #rule2.2
INPUT (REF (#ProcessorType),
       REF (#ProcessorImpl))
OUTPUT (REF (#hwProcessor));
```

The `#rule1` operation transforms a `SystemType` and its associated `SystemImpl` of an AADL model to their corresponding concepts in MARTE (`saAnalysisContext` and `gaResourcesPlatform`). The `#rule2` operation transforms `SystemClassifier` subcomponents of an AADL model to `Resouce` elements of the `gaResourcesPlatform`.

## 4.2 Definition of implementations

Once we have defined model transformation operations, we establish their associated implementations descriptions. Statements of Listing 8.5 define implementations descriptions of the operations previously defined.

Listing 8.5: A subset of OntoQL statements to define implementations of `AADL2MARTE` transformation

```
CREATE IMPLEMENTATION #rule1JavaImp
DESCRIPTORS (
    type = 'java',
    location = '193.55.../programs.jar',
    class = 'fr.ensma.lias.AadlToMarte'
    method = 'rule1Imp')
IMPLEMENTS #rule1;

CREATE IMPLEMENTATION #rule2JavaImp
DESCRIPTORS (
    ...
    method = 'rule2Imp')
IMPLEMENTS #rule2;
```

```
CREATE IMPLEMENTATION #rule2.1JavaImp
DESCRIPTORS (
    ...
    method = 'rule2.1Imp')
IMPLEMENTS #rule2.1;

CREATE IMPLEMENTATION #rule2.1JavaImp
DESCRIPTORS (
    ...
    method = 'rule2.2Imp')
IMPLEMENTS #rule2.2;
```

For instance the pseudocode of the `rule1` is given in Listing 8.6.

Listing 8.6: Pseudocode of `rule1`

```
SaAnalysisContext rule1Imp(SytemType sType, SystemImp sImp) {
    return SaAnalysisContext(sType, sImp);
}

GaResourcesPlatform rule1Imp(SytemType sType, SystemImp sImp) {
    return GaResourcesPlatform(sType, sImp);
}

SaAnalysisContext (SytemType sType, SystemImp sImp) {
    SaAnalysisContext s = new SaAnalysisContext();
    s.setName(sType.getName());
    ...
    return s;
}

GaResourcesPlatform (SytemType sType, SystemImp sImp) {
    GaResourcesPlatform g = new GaResourcesPlatform();
    g.setName(sType.getName());
    ...
    return g;
}
```

## 4.3 Exploitation of the defined operations

After defining operations and their implementations descriptions, the defined operations can be invoked in order to transform AADL concepts to MARTE ones. For instance, we can use the established operations for obtaining our AADL source model expressed in the MARTE formalism. We can also transform the same model to a MARTE model in order to have two representations of our system (Listing 8.7).

Listing 8.7: A subset of OntoQL statements to transform AADL models to MARTE

```
CREATE #saAnalysisContext marte_embSys
AS SELECT #rule1 (embSys, embSys.Impl)
   FROM #SystemClassifier;

CREATE #gaResourcesPlatform marte_embSys
AS SELECT #rule1 (embSys, embSys.Impl)
   FROM #SystemClassifier;

CREATE #MemoryPartition marte_memory
AS SELECT #rule2.1 (proc_embSys, proc_embSys.Impl)
   FROM #ProcessClassifier;
```

```
CREATE #Scheduler marte_scheduler
AS SELECT #rule2.1 (proc_embSys, proc_embSys.Impl)
   FROM #ProcessClassifier;

CREATE #gaResourcesPlatform marte_embSys_cpu
AS SELECT #rule2(embSys_cpu)
   FROM #Processor;
```

The first statement selects the MARTE system resulting from the transformation of the AADL model of our example, while the second statement reads the resulting MARTE memory and scheduler concepts from the transformation of the `proc_embSys` process element of the AADL model of our system. Whereas, the last statement creates a `gaResourcesPlatform` instance from the resulting transformation of the `embSys_cpu` processor of the AADL model. The MARTE model resulting from the transformation of the AADL model of our example (Figure 8.3) is the represented in Figure 8.5

These are only examples of the multiple transformation operations and operation invocations we have written in order to permit a complete transformation and mapping from AADL to MARTE. This eases accessing, updating, deleting and transforming AADL models even if we do not adopt AADL as a main language for design real-time and embedded system. We can also go further by setting up an operation that analyzes the schedulability of our AADL model. We detail this aspect in the next section.

## 5 Using operations for model analysis

Once the MARTE model is obtained after the transformation described in the previous section, it becomes possible to trigger scheduler analysis on these MARTE models. To do so, we define an operation whose objective is to analyze the schedulability of our system. To achieve this task, we create an operation `isSchedulable` that takes as input a MARTE model and returns as output a boolean value which states whether the systems is schedulable or not. This operation is implemented with a Java program that invokes the MAST[7] analysis tool (Listing 8.8).

Listing 8.8: Definition of the analysis operation and its implementation

```
CREATE OPERATION #isSchedulable
INPUT (REF (#MARTEModel))
OUTPUT (BOOLEAN);

CREATE IMPLEMENTATION #isSchedulableJavaImp
DESCRIPTORS (
      type = 'Java',
      location = '193.55.../programs.jar',
      class = 'fr.ensma.lias.Analyzer'
      method = 'isSchedulerImp')
IMPLEMENTS #isSchedulable;
```

The `isSchedulableJavaImp` implementation corresponds to a Java program that invokes the MAST real-time analysis tool. A part of the code of this Java program is defined in Listing 8.9.

Listing 8.9: The Java implementation of the analysis operation

[7]http://mast.unican.es/

```
public void isSchedulerImp(MarteModel model) {
    ...
    buildInputFile_mast(model);
    Runtime.getRuntime().exec("mast_analysis default -c -p -t monoprocessor
                                inputFile_mast.txt outputFile_mast.out.xml");
    ...
}
```

Now, it becomes possible to run the analysis by invoking the defined operation in the statement of Listing 8.10.

Listing 8.10: Example of an OntoQL statement to analyze an AADL model

```
SELECT #isSchedulable(#aadl2Marte(embSys.Impl))
FROM #SystemImpl
```

This statement asserts whether the `embSys` system type, transformed from AADL to MARTE by the `#aadl2marte` operation, is schedulable or not. Here, we analyze the schedulability of the corresponding MARTE model of our system. Note also that the invocation of the `isSchedulable` operation needs a MARTE model as input and thus, we provide an operation invocation as an argument of the `#isSchedulable` operation since the BeMoRe PMMS supports such manipulation. The previous statement invokes the `isSchedulable` analysis operation which is specific to MARTE, using an AADL resource hiding the transformation process to the user.

## 6 Conclusion

As a further step to validate our approach, we have shown in this chapter two applications. The first application consists in transforming AADL models, representing real-time and embedded systems, to their corresponding MARTE ones by using operations that can be implemented with external programs or web services. This application can be used to share real-time and embedded systems regardless of the language used to design them. The second application consists in defining an operation to analyze the schedulability of a real-time system. This operation is implemented by an external program that invokes a real-time models analysis tool. Moreover, we have shown through this work that different heterogeneous models associated to complex systems can be stored, manipulated, retrieved and analyzed in a persistent setting offering a hosting infrastructure for model repositories. The work presented in this chapter has been validated in [Bazhar et al., 2013b].

# Conclusions and perspectives

## Conclusion

In our work, we have defined the notion of Persistent MetaModeling System (PMMS) as a database environment dedicated to metamodeling and model management gathering, in the same setting, metamodeling and model management features together with database characteristics. Thus, a PMMS consists in (1) a database (or a model repository) to store metamodels, models and instances, and (2) an associated exploitation language to define and to manipulate different abstraction layers data stored in the database. As we have outlined, if existing PMMSs provide the capability to define metamodels, models and instances, they do not offer the same flexibility than classical MMS to define and implement model manipulations such as transformation, mapping or source code generation. For instance, some PMMSs provide a fixed set of hard-coded operators that are not always suited to achieve the desired model management tasks. Other PMMSs provide the capability to set up user-defined operations (functions and procedures). Yet, the implementation of these operations can only be achieved with a given programming language and thus, it makes it difficult to reuse existing pieces of software. Furthermore, the implementations of operations have to be stored in a special file system. Restarting the PMMS is required each time a new operation is introduced. In the work presented in this thesis, we have designed and prototyped the BeMore PMMS as an answer to these limitations. As most PMMSs only define the structural and descriptive semantics of metamodels and models, the focus of our work was to equip BeMore with various programming capabilities to express the procedural semantics. This procedural semantics is mandatory to achieve model management tasks such as model transformation or analysis. Our contributions are summarized bellow.

## Contributions

### Definition of a set of requirements for a complete PMMS

After the study of the related work, the first contribution of our thesis is the definition of a set of requirements for a complete PMMS. Indeed, as PMMSs are database environments dedicated to metamodeling and model management, they shall gather in an integrated manner metamodeling and model management features and database characteristics. Thus, we claimed that a PMMS shall evolve in a database environment offering a persistent environment for metamodeling and model management. This means that metamodels, models and instances should all be persisted in a database and managed using the associated PMMS exploitation language in order to handle metamodeling and model management tasks. A PMMS shall also support an extensible metamodeling architecture such that multiple metamodels can be defined. With this functionality, a PMMS can support different modeling languages and approaches. To define metamodels and models, the PMMS shall provide primitives to express the structural and descriptive semantics of metamodels and models in terms of conceptual elements (e.g., classes, entities, attributes, properties). Besides, a PMMS shall offer the capability to introduce dynamically model and data management operations, and to exploit the defined operations for model and data management tasks such as transformation or data integration. To ease the reuse of existing code, the implementation of these operations should be possible using different mechanisms. For instance, the classical database internal mechanisms (e.g., stored procedures, triggers) could be used as well as external mechanisms such as external programs (e.g., Java or C++ programs) or web services in order to get benefits from their coverage and their completeness. Finally, to guarantee a high availability of the PMMS, introducing a user-defined operation should not require restarting the PMMS as this may affect the robustness of the system.

## Formal extension of PMMS to support procedural semantics

To fulfill all the defined requirements, our approach consists in extending an existing PMMS to support the procedural semantics that most PMMSs do not provide. Thus, we have achieved a formal extension of PMMS related to its metametamodel and to the algebra of the PMMS exploitation language.

To support model management operations, we have extended the metametamodel supported by several PMMSs with new concepts. This extension introduced the concepts of `Operation` and `Implementation` to the PMMS metametamodel which represent respectively a model management operation, and its associated implementations. In order to guarantee the flexibility of implementations, we have associated the concept of `Descriptor` to `Implementation`. The `Descriptor` concept associates pairs of *(key, value)* elements where each element represent respectively an implementation descriptor and its value. Furthermore, we have performed the same extension at the metamodel level in order to define operations and implementations at the data level as well.

The algebra of the PMMS exploitation language has been enriched with operators to define model and data management operations. This extension consists in the *RUN* operator which invokes a given operations in a statement of the PMMS exploitation language.

In order to store operations signatures and the descriptions of their associated implementations, we have extended the logical metamodel and the logical model of the PMMS model repository with new structures (tables) corresponding to the concepts added to the PMMS metametamodel.

## Prototyping

Our proposition to extend PMMSs with procedural semantics has been implemented on the OntoDB/OntoQL PMMS. The extension of this PMMS, called *BeMoRe*, has been performed in three steps: extension of the OntoDB platform, extension of the OntoQL language and definition of the *behavior API* that bridges BeMoRe prototype and external programming environments.

The need to invoke external programs and web services from the PMMS environment raised two main issues. The first one is the difference between data types of the PMMS and those of the different external environments (type mismatch). The second problem concerned the necessity to bridge PMMSs and external environments. To address these two issues, we have set up an Application Programming Interface (API), called the *behavior API*, as a mapping between the BeMoRe data types and the data types of the external environments. Furthermore, this API invokes external programs and web services.

We have implemented the logical extension of the PMMS model repository on the OntoDB platform. This implementation added tables to store signatures of operations and descriptions of their associated implementations with metadata.

To be able to define and exploit operations within the PMMS, the PMMS exploitation language has been extended with such capabilities. The OntoQL grammar has been extended with new instructions to define model and data management operations and implementations. Moreover, invoking model and data management operations in OntoQL statements became possible.

## Use cases

To validate the proposition we make in this thesis, we have used our approaches in three use cases to show the usefulness of our proposition.

The first use case concerns the management of ontologies. Ontologies are specific domain models that are composed by primitive and derived concepts (classes and attributes) i.e., concepts defined in terms of other ones. Usually, the derived concepts, also called *non canonical concepts* are computed using semantic reasoners. During the recent years, a new type of databases, called *Ontology-Based DataBase* (OBDB) has been proposed to store and to manipulate ontologies together with the data they describe. OBDBs are specific PMMSs dedicated to ontologies. Yet, existing OBDBs provide hard-coded mechanisms to compute non canonical concepts. For instance, some OBDBs use the database procedural language to compute non canonical concepts and others use their own semantic reasoners. These solutions are not always suitable for computing any derived concept. To solve this issue, we firstly raised the need of a more flexible management of non canonical concepts in OBDBs. Then, we have defined operations to compute non canonical concepts concepts within our PMMS.

Our approach and the work accomplished in Chapter 6 have been used in the context of an OBDB design methodology. Indeed, this methodology includes a step in which non canonical concepts are identified. These non canonical concepts are computed using semantic reasoners before building the database. Yet, the update of the ontology (e.g., adding derived classes or properties) requires replaying the OBDB design process from scratch since OBDBs lack the support of flexible operators to compute non canonical concepts. We have deployed our approach in this context in order to offer the capability to compute the derived classes and properties if they are introduced even after building the OBDB.

The last use case concerns real-time and embedded systems. These systems can be designed using different languages (e.g., AADL, MARTE) that have different purposes. For instance, AADL is oriented towards the definition of the architecture of a system, while MARTE is oriented towards real-time analysis of embedded systems which is not supported by AADL. Thus, to analyze an AADL model we should go through a model transformation from AADL to MARTE. We have used our proposition firstly to show how we can establish transformation rules between AADL and MARTE models. Then, we have set up a model analysis operation to analyze MARTE models. The last step was the analysis of an AADL model through a model transformation from AADL to MARTE.

## Perspectives

Our work opens many perspectives. Some of these perspectives are related to the functional capabilities of PMMSs, and others address non functional issues. These perspectives are presented below.

### The definition of a generic procedural language for PMMS

The objective of this perspective is to define a generic imperative and object-oriented language to express the body of model and data management operations instead of the behavior API which is currently frozen. This requires the definition of the metamodel (the abstract syntax) and the concrete syntax of the language. Thus, a program expressed within this language will be an instance of the metamodel of the language. Moreover, a program could be exported to other languages (e.g., Java, C++) by defining modules to export model and data management operations to other languages. An API will be generated for each specific language defined as an instance of this metamodel. Here the challenge concerns the storage of the defined programs and the virtual machine where this program will be executed.

### Orchestration of heterogeneous operations

The purpose of this perspective is to provide the capability to define derived operations composed from other ones. If this perspective is achieved, we will be able to compose heterogeneous operations. For instance, we may have a derived operation composed of a Java operation and a web service. Yet, the difficulty to accomplish this perspective is to manage the data flow between the different operations composing the derived one. The work achieved for web services orchestration may be exploited to address this issue.

### Optimizations of our approach

In our work, our focus was more on functionalities of the proposed BeMoRe PMMS than on its scalability. Thus, further investigations need to be done in this direction. A thorough evaluation of the scalability of BeMoRe as well as its optimization is a short term perspective. As the MapReduce paradigm is a promising direction providing scalability and massively parallel processing of large-volume data, an interesting long-term perspective consists in exploring how this technology can be combined with PMMSs. An extensive performance evaluation and comparison with the existing approaches are parts of our future work as well. It will also be interesting to explore the optimizations that can be done on the PMMS query language, and to consider other non functional properties such as transactions and security.

# Bibliography

[ans, 1975] (1975). Interim report: Ansi/x3/sparc study group on data base management systems 75-02-08. *FDT - Bulletin of ACM SIGMOD*, 7(2):1–140.

[mda, 2003] (2003). Mda guide version 1.0.1. Technical report, Object Management Group.

[mof, 2011] (2011). Meta object facility (mof). Technical report, Object Management Group.

[qvt, 2011] (2011). Meta object facility (mof) 2.0 query/view/transformation specification. Technical report, Object Management Group.

[xmi, 2011] (2011). Omg mof 2 xmi mapping specification. Technical report, Object Management Group.

[mar, 2011] (2011). Uml profile for marte : Modeling and analysis of real-time embedded systems. Technical report, Object Management Group.

[uml, 2011] (2011). Unified modeling language (uml). Technical report, Object Management Group.

[aad, 2012] (2012). Architecture analysis & design language (aadl). Technical report, SAE International.

[hib, 2012] (2012). *Hibernate Reference Documentation*. The Hibernate Team and The JBoss Visual Design Team.

[acc, 2013] (2013). Acceleo.

[cdo, 2013] (2013). The cdo model repository.

[emf, 2013a] (2013a). Eclipse modeling framework project (emf).

[ecl, 2013] (2013). Eclipselink.

[emf, 2013b] (2013b). *EMF Store*. The Eclipse Foundation. http://www.eclipse.org/proposals/emf-store/.

[sta, 2013] (2013). Staruml - the open source uml/mda platform.

[ten, 2013] (2013). *Teneo*. The Eclipse Foundation.

[top, 2013] (2013). Topcased.

[Alexaki et al., 2001] Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D., and Tolle, K. (2001). The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In *Proceedings of the 2nd International Workshop on the Semantic Web*, pages 1–13.

[Bazhar, 2012] Bazhar, Y. (2012). Handling behavioral semantics in persistent meta-modeling systems. In *RCIS*, pages 1–6.

[Bazhar et al., 2013a] Bazhar, Y., Ameur, Y. A., and Jean, S. (2013a). Bemore: a repository for handling models behaviors. In *SEKE*.

[Bazhar et al., 2012a] Bazhar, Y., Ameur, Y. A., Jean, S., and Baron, M. (2012a). A flexible support of non canonical concepts in ontology-based databases. In *WEBIST*, pages 393–398.

[Bazhar et al., 2012b] Bazhar, Y., Chakroun, C., Ameur, Y. A., Bellatreche, L., and Jean, S. (2012b). Extending ontology-based databases with behavioral semantics. In *OTM Conferences (2)*, pages 879–896.

[Bazhar et al., 2013b] Bazhar, Y., Ouhammou, Y., Ameur, Y. A., Grolleau, E., and Jean, S. (2013b). Persistent meta-modeling systems as heterogeneous model repositories. In *MEDI*.

[Bellatreche et al., 2011] Bellatreche, L., Ameur, Y. A., and Chakroun, C. (2011). A design methodology of ontology based database applications. *Logic Journal of the IGPL*, 19(5):648–665.

[Bernstein et al., 1999] Bernstein, P. A., Bergstraesser, T., Carlson, J., Pal, S., Sanders, P., and Shutt, D. (1999). Microsoft repository version 2 and the open information model. *Inf. Syst.*, 24(2):71–98.

[Borgida and Brachman, 1993] Borgida, A. and Brachman, R. J. (1993). Loading data into description reasoners. *SIGMOD Record*, 22(2):217–226.

[Brickley and Guha, 2004] Brickley, D. and Guha, R. V. (2004). *RDF Vocabulary Description Language 1.0: RDF Schema*. World Wide Web Consortium. http://www.w3.org/TR/rdf-schema.

[Broekstra et al., 2002] Broekstra, J., Kampman, A., and van Harmelen, F. (2002). Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In Horrocks, I. and Hendler, J., editors, *Proceedings of the 1st International Semantic Web Conference (ISWC'02)*, number 2342 in Lecture Notes in Computer Science, pages 54–68. Springer Verlag.

[Cabot and Gogolla, 2012] Cabot, J. and Gogolla, M. (2012). Object constraint language (ocl): A definitive guide. In *SFM*, pages 58–90.

[Calbimonte et al., 2009] Calbimonte, J.-P., Porto, F., and Keet, C. M. (2009). Functional dependencies in owl abox. In *SBBD*, pages 16–30.

[Carroll et al., 2004] Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., and Wilkinson, K. (2004). Jena: Implementing the Semantic Web Recommendations. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters (WWW'04)*, pages 74–83, New York, NY, USA. ACM Press.

[Chakroun et al., 2011] Chakroun, C., Bellatreche, L., and Ameur, Y. A. (2011). The role of class dependencies in designing ontology-based databases. In *OTM Workshops*, pages 444–453.

[Chong et al., 2005] Chong, E. I., Das, S., Eadon, G., and Srinivasan, J. (2005). An Efficient SQL-based RDF Querying Scheme. In *Proceedings of the 31st international conference on Very Large Data Bases (VLDB'05)*, pages 1216–1227.

[Clasen et al., 2012] Clasen, C., Didonet Del Fabro, M., and Tisi, M. (2012). Transforming very large models in the cloud: a research roadmap. In *First International Workshop on Model-Driven Engineering on and for the Cloud*, Copenhagen, Denmark. Springer.

[Das et al., 2004] Das, S., Chong, E. I., Eadon, G., and Srinivasan, J. (2004). Supporting ontology-based semantic matching in rdbms. In *VLDB*, pages 1054–1065.

[Dean and Schreiber, 2004] Dean, M. and Schreiber, G. (2004). *OWL Web Ontology Language Reference*. World Wide Web Consortium. http://www.w3.org/TR/owl-ref.

[Dehainsala et al., 2007] Dehainsala, H., Pierra, G., and Bellatreche, L. (2007). Ontodb: An ontology-based database for data intensive applications. In *DASFAA Conference*.

[Espinazo-Pagán et al., 2011] Espinazo-Pagán, J., Cuadrado, J. S., and Molina, J. G. (2011). Morsa: A scalable approach for persisting and accessing large models. In *MoDELS*, pages 77–92.

[Grant et al., 1993] Grant, J., Litwin, W., Roussopoulos, N., and Sellis, T. (1993). Query languages for relational multidatabases. *The VLDB Journal*, 2:153–172.

[Gruber, 1993] Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220.

[Gruber, 1995] Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies (IJHCS)*, 43(5-6):907–928.

[Guy et al., 2003] Guy, P., Yamine, A.-A., and Eric, S. (2003). ISO (660p), GenÃ¨ve.

[Harris and Gibbins, 2003] Harris, S. and Gibbins, N. (2003). 3store: Efficient bulk RDF Storage. In *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PPP'03)*, pages 1–15.

[Hayes, 2004] Hayes, P. (2004). *RDF Semantics*. World Wide Web Consortium. http://www.w3.org/TR/rdf-mt/.

[Hernández et al., 2001] Hernández, M. A., Miller, R. J., and Haas, L. M. (2001). Clio: a semi-automatic tool for schema mapping. In *SIGMOD Conference*.

[Hick and Hainaut, 2003] Hick, J.-M. and Hainaut, J.-L. (2003). Strategy for database application evolution: The db-main approach. In *ER*, pages 291–306.

[Jarke et al., 2009a] Jarke, M., Jeusfeld, M. A., Nissen, H. W., and Quix, C. (2009a). Heterogeneity in model management: A meta modeling approach. In *Conceptual Modeling: Foundations and Applications*, pages 237–253.

[Jarke et al., 2009b] Jarke, M., Jeusfeld, M. A., Nissen, H. W., Quix, C., and Staudt, M. (2009b). Metamodelling with datalog and classes: Conceptbase at the age of 21. In *ICOODB*, pages 95–112.

[Jean et al., 2006a] Jean, S., Ameur, Y. A., and Pierra, G. (2006a). Querying ontology based database using ontoql (an ontology query language). In *OTM Conferences (1)*, pages 704–721.

[Jean et al., 2006b] Jean, S., Ameur, Y. A., and Pierra, G. (2006b). Querying ontology based databases - the ontoql proposal. In *SEKE*, pages 166–171.

[Jean et al., 2007] Jean, S., Ameur, Y. A., and Pierra, G. (2007). An object-oriented based algebra for ontologies and their instances. In *ADBIS*, pages 141–156.

[Jean et al., 2006c] Jean, S., Pierra, G., and Ameur, Y. A. (2006c). Domain ontologies: A database-oriented analysis. In *WEBIST (Selected Papers)*, pages 238–254.

[Jeusfeld et al., 2013] Jeusfeld, M. A., Quix, C., and Jarke, M. (2013). *ConceptBase .cc User Manual*. Tilburg University, RWTH Aachen.

[Jézéquel et al., 2009] Jézéquel, J.-M., Barais, O., and Fleurey, F. (2009). Model driven language engineering with kermeta. In *GTTSE*, pages 201–221.

[Jouault and Kurtev, 2005] Jouault, F. and Kurtev, I. (2005). Transforming models with atl. In *MoDELS Satellite Events*, pages 128–138.

[Jouault and Tisi, 2010] Jouault, F. and Tisi, M. (2010). Towards incremental execution of atl transformations. In Tratt, L. and Gogolla, M., editors, *ICMT2010 - Intl. Conference on Model*

*Transformation*, volume 6142 of *Lecture Notes in Computer Science*, pages 123–137, Malaga, Spain. Springer.

[Kensche et al., 2007] Kensche, D., Quix, C., Chatti, M. A., and Jarke, M. (2007). Gerome: A generic role based metamodel for model management. *J. Data Semantics*, 8:82–117.

[Koegel and Helming, 2010] Koegel, M. and Helming, J. (2010). Emfstore : a model repository for emf models. In *ICSE (2)*, pages 307–308.

[Kolovos et al., 2013] Kolovos, D., Rose, L., GarcÃa-DomÃnguez, A., and Paige, R. (2013). *The epsilon Book*.

[Lu et al., 2007] Lu, J., Ma, L., Zhang, L., Brunner, J.-S., Wang, C., Pan, Y., and Yu, Y. (2007). Sor: a practical system for ontology storage, reasoning and search. In *vldb*, pages 1402–1405.

[Mallet et al., 2009] Mallet, F., André, C., and DeAntoni, J. (2009). Executing aadl models with uml/marte. In *ICECCS*, pages 371–376.

[Manola and Miller, 2004] Manola, F. and Miller, E. (2004). *RDF Primer*. World Wide Web Consortium. http://www.w3.org/TR/rdf-primer.

[Mei et al., 2006] Mei, J., Ma, L., and Pan, Y. (2006). Ontology query answering on databases. In *Proceedings of the 5th International Semantic Web Conference (ISWC'06)*, pages 445–458.

[Melnik et al., 2003] Melnik, S., Rahm, E., and Bernstein, P. A. (2003). Rondo: A programming platform for generic model management. In *SIGMOD Conference*, pages 193–204.

[Minsky, 1968] Minsky, M. (1968). The society of mind.

[Mylopoulos et al., 1990] Mylopoulos, J., Borgida, A., Jarke, M., and Koubarakis, M. (1990). Telos: Representing knowledge about information systems. *ACM Trans. Inf. Syst.*, 8(4):325–362.

[Paige et al., 2011] Paige, R. F., Kolovos, D. S., Rose, L. M., Matragkas, N. D., and Williams, J. R. (2011). Model management in the wild. In *GTTSE*, pages 197–218.

[Pan and Heflin, 2003] Pan, Z. and Heflin, J. (2003). DLDB: Extending Relational Databases to Support Semantic Web Queries. In *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03)*, pages 109–113.

[Park et al., 2007] Park, M. J., Lee, J. H., Lee, C. H., Lin, J., Serres, O., and Chung, C. W. (2007). An Efficient and Scalable Management of Ontology. In *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07)*, volume 4443 of *Lecture Notes in Computer Science*, pages 975–980. Springer.

[Petrini and Risch, 2007] Petrini, J. and Risch, T. (2007). SWARD: Semantic Web Abridged Relational Databases. In *Proceedings of the 18th International Conference on Database and Expert Systems Applications (DEXA'07)*, pages 455–459.

[Pierra, 2007] Pierra, G. (2007). Context Representation in Domain Ontologies and its Use for Semantic Integration of Data. *Journal Of Data Semantics (JODS)*, X:34–43.

[Pierra and Sardet, 2010] Pierra, G. and Sardet, E. (2010). *ISO 13584-32 - Industrial automation systems and integration - Parts library - Part 32: Implementation resources: OntoML: Product ontology markup language*. ISO.

[Pierra et al., 1995] Pierra, G., Sardet, E., and Withier, R. (1995). Modélisation des données : le langage express. Technical report, ENSMA.

[Romero et al., 2009] Romero, O., Calvanese, D., Abelló, A., and Rodriguez-Muro, M. (2009). Discovering functional dependencies for multidimensional design. In *DOLAP*, pages 1–8.

[Sirin and Parsia, 2004] Sirin, E. and Parsia, B. (2004). Pellet: An owl dl reasoner. In *Description Logics*.

[Stocker and Smith, 2008] Stocker, M. and Smith, M. (2008). Owlgres: A scalable owl reasoner. In *The Sixth International Workshop on OWL: Experiences and Directions*.

[Volz et al., 2005] Volz, R., Staab, S., and Motik, B. (2005). Incrementally Maintaining Materializations of Ontologies Stored in Logic Databases. *Journal of Data Semantics II*, 3360:1–34.

# List of Figures

# List of Tables

# Listings

# Related publications

- *Youness BAZHAR*, *Yassine OUHAMMOU*, *Yamine AIT-AMEUR*, *Emmanuel GROLLEAU*, *Stéphane JEAN*, **Persistent Meta-Modeling Systems as Heterogeneous Model Repositories**, the 3rd International Conference on Model and Data Engineering (MEDI), Amantea-Italy, September 2013.
- *Youness BAZHAR*, *Yamine AIT-AMEUR*, *Stéphane JEAN*, **BeMoRe: a Repository for Handling Models Behaviors**, the 25th International Conference on Software Engineering and Knowledge Engineering (SEKE), Boston(MA)-USA, June 2013.
- *Youness BAZHAR*, *Chedlia CHAKROUN*, *Yamine AIT-AMEUR*, *Ladjel BELLATRECHE*, *Stéphane JEAN*, **Extending Ontology-Based Databases with Behavioral Semantics**, the 11th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE), Rome-Italy, September 2012.
- *Youness BAZHAR*, **Handling Behavioral Semantics in Persistent Meta-Modeling Systems**, the 6th IEEE International Conference on Research Challenges in Information Science (RCIS), Valencia-Spain, May 2012 (*best doctoral paper*).
- *Youness BAZHAR*, *Yamine AIT-AMEUR*, *Stéphane JEAN*, *Mickael BARON*, **A Flexible Support of Non Canonical Concepts In Ontology-Based Databases**, the 8th International Conference on Web Information Systems and Technologies (WEBIST), Porto-Portugal, April 2012.
- *Youness BAZHAR*, *Stéphane JEAN*, *Yamine AIT-AMEUR*, *Mickael BARON*, **Extension de ONTODB pour construire une architecture générique de bases de données à base ontologique**, 6ème Conférence francophone sur les Architectures Logicielles (CAL), Montpellier-France, May 2012.

# Résumé étendu

## Contexte

Depuis leur apparition, les bases de données sont en constante évolution. Même si les bases de données relationnelles demeurent celles qui sont jusqu'à présent les plus utilisées, l'émergence de différents paradigmes a entraîné le développement d'autres types de base de données. Ainsi, lorsque le paradigme orienté objet est apparu, les bases de données relationnel-objet et les bases de données orientées objet ont été introduites pour supporter les particularités de ce formalisme comme par exemple la relation d'héritage. Nous pouvons constater la même tendance avec l'émergence d'autres types de bases de données tels que les bases de données XML, les bases de données orientées document ou les bases de données orientées graphe. Aujourd'hui, la notion de modèle est utilisée plus que jamais en génie logiciel et les modèles produits sont de plus en plus volumineux. Ainsi, une nouvelle génération de bases de données, appelées bases de données de modèles a vu le jour. Ce nouveau type de base de données est dédié au stockage des méta-modèles, des modèles et des instances. L'objectif principal de notre thèse est d'étendre ces bases de données de modèles avec des fonctionnalités avancées. Mais avant de présenter plus précisément nos objectifs, nous motivons en détail, dans ce qui suit, la nécessité de cette nouvelle génération de bases de données.

La conception des systèmes complexes utilise souvent différentes méthodologies, plateformes, langages, etc. Pour faire face à ce défi, une méthodologie de développement logiciel, appelée l'Ingénierie Dirigée par les Modèles (IDM), a été proposée. Dans cette méthodologie, la conception des systèmes complexes est basée sur l'utilisation d'un ensemble de modèles hétérogènes. Ces modèles représentent un système suivant différent points de vue. Par exemple, la modélisation et l'analyse des systèmes embarqués s'appuient sur différents langages de modélisation tels que SysML, UML/MARTE ou AADL. L'exploitation des modèles utilisés pour concevoir un système est basée sur des analyses telles que la génération du code source du logiciel ou la transformation d'un modèle dans un autre langage. L'objectif de l'IDM est ainsi d'améliorer le processus de développement logiciel et de faciliter la maintenance des logiciels en découplant la conception d'un système de sa mise en oeuvre.

L'IDM a suscité un grand d'intérêt dans l'industrie grâce aux avantages qu'elle offre. L'IDM est largement utilisée dans divers domaines tels que l'aéronautique ou l'automobile et contribue à la création de différents types d'applications telles que les bases de données, les langages dédiés (DSL), ou les systèmes d'information. En outre, l'IDM est utilisée pour développer des solutions d'intégration de systèmes et d'échange de données. Le succès de l'IDM a conduit à l'élaboration de plusieurs normes, outils, langages, etc. Par exemple, l'OMG (Object Management Group) a proposé l'architecture standard MDA (Model-Driven Architecture) comme une spécialisation (c'est-à-dire une mise en oeuvre) de l'IDM. MDA est une méthodologie de développement logiciel développée autour d'un ensemble de normes (par exemple, le MOF ou le langage UML), langages (par exemple, ATL ou OCL) et outils (par exemple, Eclipse Modeling Framework, Acceleo).

Avec l'utilisation croissante de l'IDM, l'exploitation des modèles volumineux (de très grande taille) est apparue comme l'un des plus grand défis rencontrés. En effet, des modèles surdimensionnés sont produits dans des domaines tels que le commerce électronique. Par exemple, les modèles peuvent contenir des millions d'éléments (par exemple, lorsqu'ils résultent de la rétro-ingénierie de grands systèmes) et peuvent décrire des données de grande taille (par exemple, Uniprot est une base de connaissance en génomique qui regroupe plus de 200 Go de séquences de protéines). Comme la plupart des systèmes de méta-modélisation (MMS) ont été initialement conçus pour une exploitation de modèles en mémoire centrale, les MMS ont certaines difficultés à gérer des modèles de grande taille. Pour résoudre le problème du passage à l'échelle de l'IDM, la plupart des approches proposées visent à améliorer le passage à l'échelle des outils d'exploitation des modèles en mémoire centrale au travers des approches qui consistent à :

- effectuer une exploitation progressive et partielle des grands modèles. L'idée est de charger, en mémoire centrale, la partie requise d'un grand modèle uniquement. Bien sûr, cette approche ne peut être suivie si une opération s'applique à l'ensemble du modèle;
- effectuer l'exploitation des grands modèles dans un contexte distribué. L'idée consiste à décomposer une tâche d'exploitation en sous-tâches. Cette approche est limitée aux opérations qui peuvent être décomposées en sous-tâches indépendantes;
- équiper les systèmes de méta-modélisation avec des bases de données dédiées pour stocker les méta-modèles, les modèles et les instances. Ces bases de données sont appelés bases de modèles (par exemple, EMFStore). Cependant, cette approche présente deux inconvénients majeurs : (i) les bases de modèles sont équipées de langages limités seulement à l'interrogation, de sorte que (ii) toutes les tâches d'exploitation de modèles (transformation, génération de code, etc.) nécessitent le chargement des modèles en mémoire principale pour qu'ils soient traités.

Contrairement à d'autres paradigmes (comme XML ou le paradigme orienté objet), peu de travaux ont été menés pour étendre les bases de données relationnelles pour gérer des modèles et méta-modèles comme des entités de première classe. ConceptBase, OntoDB/OntoQL et Rondo sont des exemples de cette approche qui consiste à définir des systèmes de méta-modélisation et d'exploitation de modèles évoluant complètement dans un environnement de base de données. Ces systèmes, appelés systèmes de méta-modélisation persistants (PMMS), consistent en (1) une base de modèles qui stocke des méta-modèles, des modèles et des instances tout en respectant la séparation entre les différents couches de métadonnées et la préservation de la conformité aux différents niveaux d'abstraction, et (2) un langage d'exploitation associé capable de créer et de manipuler les méta-modèles, les modèles et les données. Les PMMS ont été proposés pour :

- bénéficier des capacités de passage à l'échelle des bases de données et en particulier de l'optimisation des requêtes faites dans les Système de Gestion de Base de Données (SGBD);

- offrir une plateforme commune pour le partage des modèles et des données puisque les SGBD fournissent des mécanismes pour sécuriser l'accès aux données partagées;

- fournir un langage déclaratif permettant de définir des modèles conformes aux différents méta-modèles.

Si les PMMS existants exploitent les caractéristiques des SGBD, ils n'offrent pas les mêmes fonctionnalités que les MMS classiques. En effet, dans la littérature, nous constatons que les PMMS proposés supportent principalement la définition de la sémantique structurelle et descriptive des modèles en fournissant des constructeurs de (méta-)classes, de (méta-)attributs ainsi que des primitives pour exprimer les relations d'héritage et d'association. Par contre, les PMMS offrent des capacités limitées pour définir des fonctions et des procédures (sémantique comportementale) qui pourraient être utiles pour accomplir des tâches comme la transformation de modèles ou la génération de code. Actuellement, les PMMS existants utilisent des langages procéduraux de base de données (par exemple, PL/SQL) qui ne supportent pas la manipulation des concepts de haut niveau (par exemple, les classes), ou fournissent un ensemble d'opérateurs prédéfinis (par exemple Compose, Match, Merge) pour effectuer des opérations très spécifiques sur les modèles. Jusque là, ConceptBase reste le PMMS le plus complet car il supporte la notion de fonction. En effet, une fonction dans ConceptBase peut être implémentée en utilisant un langage déductif imposé. Par contre, ce système ne permet ni d'utiliser des programmes externes implémentés dans d'autres langues (par exemple, Java), ni des services web. Par ailleurs, ConceptBase requiert le redémarrage du PMMS à chaque fois qu'une nouvelle fonction externe est introduite, ce qui limite la disponibilité des PMMS (démarrage à froid).

## Notre proposition

Le but de notre travail est de concevoir un PMMS qui rassemble les caractéristiques fonctionnelles des MMS classiques et les avantages des SGBD. Notre approche vise ainsi à définir un SGBD permettant de stocker des méta-modèles, des modèles et des instances qui soit associé à un langage d'exploitation déclaratif. Ce langage doit offrir la possibilité de définir, manipuler (mettre à jour, supprimer et interroger) ces différents niveaux d'abstraction. De plus, ce PMMS devrait permettre de réaliser des opérations avancées sur les modèles comme la transformation de modèles ou le calcul des concepts dérivés en utilisant des mécanismes procéduraux flexibles tels que des programmes externes (par exemple, Java, C++) ou des services Web.

Pour atteindre cet objectif, notre travail se propose d'utiliser une plateforme de méta-modélisation persistante. Plus précisément, la solution que nous proposons est basée sur le PMMS OntoDB/OntoQL mais elle peut être appliquée à d'autres PMMS. Si le PMMS OntoDB/OntoQL supporte la sémantique structurelle et descriptive des modèles et des méta-modèles à travers son langage associé (OntoQL), la définition du comportement des éléments de modèles n'est pas encore supportée. Par conséquent, ce système n'est pas complet et doit être étendu afin de supporter la sémantique comportementale (par exemple, les opérations, les contraintes, les expressions, les dérivations, etc.). Comme ce système supporte la manipulation des modèles comme des objets de première classe grâce à ses capacités de méta-modélisation, cette extension permettra d'enrichir le PMMS OntoDB/OntoQL en permettant différents traitements sur les modèles tels que des transformations de modèles, l'intégration de données, la vérification de contraintes, etc. Dans cette thèse, notre travail contribue à :

- la définition d'un ensemble d'exigences pour un PMMS complet et un état de l'art qui montre que les PMMS existants ne remplissent pas les exigences définies;
- la définition formelle d'un modèle de données des PMMS et du langage d'exploitation associé supportant la sémantique comportementale;
- la mise en oeuvre de notre proposition (le PMMS BeMoRe) avec quelques expérimentations préliminaires sur ses capacités pour le passage à l'échelle;
- le développement de trois cas d'utilisation complets dans différents domaines pour montrer la validité et l'utilité de notre démarche.

## Structure du mémoire

Cette thèse est structurée comme suit.

- Le **Chapitre 1** introduit la notion de l'IDM, y compris les concepts de métamodélisation et l'exploitation des modèles qui sont d'un intérêt particulier dans cette thèse.
- Le **Chapitre 2** présente les solutions de persistance existantes dédiées à la métamodélisation et à l'exploitation des modèles. Nous décrivons d'abord les approches utilisées par les MMS classiques pour surmonter le problème du passage à l'échelle. Ensuite, nous introduisons la notion de système de métamodélisation persistant (PMMS), sur lequel nous concentrons notre travail, et nous discutons les avantages et les limites des PMMS existants.
- Le **Chapitre 3** définit les exigences d'un PMMS complet. Ces exigences incluent les différentes caractéristiques de métamodélisation et d'exploitation des modèles ainsi que les avantages des systèmes de base de données (par exemple, le passage à l'échelle, les capacités d'interrogation). Enfin, nous analysons les PMMS existants en fonction des exigences définis.
- Le **Chapitre 4** expose l'extension formelle du métamétamodèle des PMMS avec de nouveaux concepts pour supporter la définition d'opérations pour l'exploitation des modèles. Ces opérations peuvent être mis en oeuvre en utilisant des mécanismes flexibles tels que les programmes externes ou les services web. Ensuite, nous introduisons l'extension formelle de l'algèbre du langage d'exploitation des PMMS avec les opérateurs de définition et d'exploitation des opérations.
- Le **Chapitre 5** présente l'extension de la syntaxe du langage OntoQL avec de nouvelles instructions pour permettre la définition d'opérations. Par ailleurs, ce chapitre présente les aspects techniques de la mise en oeuvre de notre approche, en particulier le processus d'exécution des instructions OntoQL comportant des invocations d'opérations. En outre, ce chapitre propose une petite étude du passage à l'échelle de notre approche qui fait partie des perspectives de notre travail.

- Le **Chapitre 6** présente un premier cas d'utilisation de notre approche. Cette étude de cas porte sur l'exploitation des concepts non canoniques (dérivés) dans les bases de données à base ontologique (BDBOs) qui sont des bases de données dédiées au stockage des ontologies.
- Le **Chapitre 7** présente un deuxième cas d'utilisation de notre proposition pour améliorer une méthodologie de conception BDBOs.
- Le **Chapitre 8** traite un cas d'utilisation de notre approche qui consiste à utiliser les opérations pour effectuer des transformations et des analyses de modèles temps réel au sein du PMMS.
- Enfin, nous listons les conclusions de notre proposition et nous exposons les travaux en cours et les perspectives ouvertes par notre travail.

# Abstract

During the recent years, modeling and model management have taken a great interest in software development since they accelerate the software development process and facilitate their maintenance. But, with the increasing size of models and their instances, especially in industrial contexts, the management of models and their instances with tools evolving in main memory presents some insufficiencies related to scalability. Indeed, classical modeling and model management tools using the central memory have showed their limits when they face large scale models and instances. Thus, to overcome the problem of scalability the management of models in databases becomes a necessity. Indeed, two solutions were proposed. The first one consists in equipping modeling and model management tools with specific databases called model repositories (e.g., EMFStore) dedicated to store metamodels, models and instances. These model repositories are equipped with exploitation languages restricted only to querying capabilities such that model repositories serve only as model warehouses as processing model management tasks require loading the whole model to the central memory. The second solution, on which we focus our approach, consists in defining database environments for metamodeling and model management. These systems, called Persistent metamodeling Systems (PMMSs), aim at providing a database environment for metamodeling and model management. Indeed, a PMMS consists in (i) a database that stores metamodels, models their instances while respecting the separation of the different abstraction layers, and (ii) an associated exploitation language possessing metamodeling and model management capabilities. Several PMMSs have been proposed (e.g., ConceptBase, OntoDB/OntoQL) and focus mainly on the structural definition of metamodels and models in terms of (meta-)classes, (meta-)attributes, etc. Yet, existing PMMSs provide limited capabilities to define behavioral semantics for model and data management. Indeed, behavioral semantics could be useful to compute derivations, perform model transformations, generate source code, etc. In our work, we propose to extend PMMSs with the capability to introduce dynamically user-defined model and data management operations. These operations can be implemented using flexible and heterogeneous mechanisms. Indeed, they can use internal database mechanisms (e.g., stored procedures, views) as well as external mechanisms such as web services or external programs (e.g., Java, C++). As a consequence, this extension enhances PMMSs giving them more coverage and further flexibility. This extension has been implemented on the OntoDB/OntoQL prototype, and experimented to check the scaling of our approach. Moreover, our proposition has been used in three different contexts. In particular, (1) to compute derived concepts of ontologies, (2) to enhance an ontology-based database design methodology and (3) to transform and analyze models of real-time and embedded systems.

**Keywords :** metamodeling, model management, structural and descriptive semantics, behavioral semantics, model repository, model persistence.

# Handling Behavioral Semantics in Persistent Metamodeling Systems

# Extension des Systèmes de Métamodélisation Persistants avec la Sémantique Comportementale

Présentée par : **Youness BAZHAR**

Directeur de Thèse : **Yamine AIT-AMEUR**
Co-encadrant : **Stéphane JEAN**

**Abstract.** Modeling and model management have taken a great interest in software development since they accelerate the software development process and facilitate their maintenance. But, with the increasing size of models and their instances, the management of models and their instances with tools evolving in main memory presents some insufficiencies related to scalability. Indeed, classical tools using the central memory have shown their limits when they face large scale models and instances. Thus, to overcome the problem of scalability, the management of models in databases becomes a necessity. Indeed, two solutions were proposed. The first one consists in equipping modeling and model management tools with specific databases, called model repositories, (e.g., EMFStore) dedicated to store metamodels, models and instances. These model repositories are equipped with exploitation languages restricted only to querying capabilities such that model repositories serve only as model warehouses as processing model management tasks require loading the whole model to the central memory. The second solution, on which we focus our approach, consists in defining database environments for metamodeling and model management. These systems, called Persistent MetaModeling Systems (PMMSs), aim at providing a database environment for metamodeling and model management. Indeed, a PMMS consists in (i) a database that stores metamodels, models their instances, and (ii) an associated exploitation language possessing metamodeling and model management capabilities. Several PMMSs have been proposed (e.g., ConceptBase, OntoDB/OntoQL) and focus mainly on the structural definition of metamodels and models in terms of (meta-)classes, (meta-)attributes, etc. Yet, existing PMMSs provide limited capabilities to define behavioral semantics for model and data management. Indeed, behavioral semantics could be useful to compute derivations, perform model transformations, generate source code, etc. In our work, we propose to extend PMMSs with the capability to introduce dynamically user-defined model and data management operations. These operations can be implemented using flexible and heterogeneous mechanisms. Indeed, they can use internal database mechanisms (e.g., stored procedures) as well as external mechanisms such as web services or external programs (e.g., Java, C++). As a consequence, this extension enhances PMMSs giving them more coverage and further flexibility. This extension has been implemented on the OntoDB/OntoQL prototype, and experimented to check the scaling of our approach. Moreover, our proposition has been used in three different contexts. In particular, (1) to compute derived concepts of ontologies, (2) to enhance an ontology-based database design methodology and (3) to transform and analyze models of real-time and embedded systems.
**Keywords:** meta-modeling, model management, structural and descriptive semantics, behavioral semantics, model repository, model persistence.

**Résumé.** L'Ingénierie Dirigée par les Modèles (IDM) a suscité un grand intérêt grâce aux avantages qu'elle offre. En particulier, l'IDM vise à accélérer le processus de développement et à faciliter la maintenance des logiciels. Mais avec l'augmentation permanente de la taille des modèles et de leurs instances, l'exploitation des modèles et de leurs instances, en utilisant des outils classiques présente des insuffisances liées au passage à l'échelle. L'utilisation des bases de données est une des solutions proposées pour répondre à ce problème. Dans ce contexte, deux approches ont été proposées. La première consiste à équiper les outils de modélisation avec des bases de données dédiées au stockage de modèles, appelées *model repositories* (p. ex. EMFStore). Ces bases de données sont équipées de langages d'exploitation limités seulement à l'interrogation des modèles et des instances. Par conséquent, ces langages n'offrent aucune capacité pour effectuer des opérations avancées sur les modèles telles que la transformation de modèles ou la génération de code. La deuxième approche, que nous suivons dans notre travail, consiste à définir des environnements persistants en base de données dédiés à la méta-modélisation. Ces environnements sont appelés *systèmes de méta-modélisation persistants* (PMMS). Un PMMS consiste en (i) une base de données dédiée au stockage des méta-modèles, des modèles et de leurs instances, et (ii) un langage d'exploitation associé possédant des capacités de méta-modélisation et d'exploitation des modèles. Plusieurs PMMS ont été proposés tels que ConceptBase ou OntoDB/OntoQL. Ces PMMS supportent principalement la définition de la sémantique structurelle et descriptive des méta-modèles et des modèles en terme de (méta-)classes, (méta-)attributs, etc. Par contre, ces PMMS fournissent des mécanismes limités pour définir la sémantique comportementale nécessaire à l'exploitation des modèles et des instances. En effet, la sémantique comportementale pourrait être utile pour calculer des concepts dérivés, effectuer des transformations de modèles, générer du code source, etc. Ainsi, nous proposons dans notre travail d'étendre les PMMS avec la possibilité d'introduire dynamiquement des opérations qui peuvent être implémentées en utilisant des mécanismes hétérogènes. Ces opérations peuvent ainsi utiliser des mécanismes internes au système de gestion de base de données (p. ex. les procédures stockées) tout comme des mécanismes externes tels que les services web ou les programmes externes (p. ex. Java, C++). Cette extension permet d'améliorer les PMMS en leur donnant une plus large couverture de fonctionnalités et une plus grande flexibilité. Pour valider notre proposition, elle a été implémentée sur le prototype OntoDB/OntoQ et a été mise en œuvre dans trois contextes différents : (1) pour calculer les concepts dérivés dans les bases de données à base ontologique, (2) pour améliorer une méthodologie de conception de base de données à base ontologique et finalement (3) pour faire de la transformation et de l'analyse des modèles des systèmes embarqués temps réel.
**Mots clés :** méta-modélisation, model management, sémantique structurelle et descriptive, sémantique comportementale, base de données, persistance des modèles.