universität wien

MASTERARBEIT

Titel der Masterarbeit

„Validation of ADOxx Metamodels based on Semantic Technologies“

verfasst von

Christos Lekaditis

angestrebter akademischer Grad

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2014

Studienkennzahl lt. Studienblatt:	A 066 926
Studienrichtung lt. Studienblatt:	Masterstudium Wirtschaftsinformatik
Betreut von:	Univ.-Prof. Dr. Prof. h.c. Dimitris Karagiannis

Thanks

I would like to thank Prof. Karagiannis, who throughout the course of my studies showed the relationship that an academic teacher should have with his student. I would also like to give my sincere thanks to my supervisor and colleague Mag. Srdjan Živković. Without his contribution, his willingness to share his scientific background with me and his moral support this work would never have managed to escape from its strict technological framework, nor reach its interdisciplinary potential.

I would also like to thank my friends, the ones that I left back in Greece, and the new ones that I made here in Vienna, who were always there for me, in good and bad times.

Finally, I would like to thank my parents, who all these three years have been trying to do their best for me. During my long journey through academia, they have always been there discretely for me to support me, advise me and keep my spirits high. Thank you.

Table of Contents

List of Figures

List of Tables

Zusammenfassung

Diese Diplomarbeit erforscht die Verwendung der semantischen Technologie als Mittel um die Korrektheit der Metamodelle zu prüfen. Ein gegebenes Metamodell soll mit seinem Meta-Metamodell konform gehen, genauso wie ein Modell mit seinem Metamodell konform geht. Die Anpassung der Metamodelle wird durch ihre syntaktisch und semantisch korrekte Definition erreicht. Mit anderen Worten - ein Metamodell soll immer in Übereinstimmung mit den Regeln, die durch die Syntax gesetzt werden, und mit dem Sinn, der durch die Semantik definiert wird, sein. Obwohl die heutigen Metamodellierungswerkzeuge syntaktische Checks unterstützen, werden die semantischen Checks nicht immer abgedeckt. Die Metamodellierungswerkzeuge, die semantische Checks unterstützen, können unter einer kleinen Anzahl von verschiedenen Ansätzen für die Umsetzung auswählen. Die Definition von semantischen Checks kann entweder hartkodiert sein oder mit der Verwendung einer Nebenbedingungssprache, wie zum Beispiel OCL (Object Constraint Language), erreicht werden. Unabhängig von dem Ansatz ist die Definition der semantischen Nebenbedingungen eine komplexe und zeitaufwendige Aufgabe. Das Hauptproblem liegt in der wachsenden Anzahl der Nebenbedingungen beim Meta-Metamodell, welche zu dem wachsenden Risiko führt, widersprüchliche Nebenbedingungen zu definieren. In dieser Arbeit werde ich die Möglichkeiten der Nutzung von semantischen Technologien als Mittel zur Definition von semantischen Nebenbedingungen erkunden. Genauer gesagt, werde ich die Möglichkeiten der semantischen Reasoners und der vielversprechenden Ontologiesprache OWL 2 (Web Ontology Language 2) - eine sich selbst beschreibende und ausdrucksstarke Sprache, die auf semantischen Technologien basierend ist - untersuchen und, ob sie etwas zu dem Gebiet der Validierung der Metamodelle beiträgt. Die Anwendbarkeit dieses Konzepts wird auf dem Meta-metamodell der ADOxx Metamodellierungsplattform basieren. Das Hauptziel dieser Arbeit ist es, eine einfachere und deklarativere Definition von semantischen Nebenbedingungen innerhalb des Meta-Metamodells zu erreichen und damit eine effizientere, schnellere und genauere Validierung der Metamodelle zu erschaffen.

FACHGEBIET: Validierung von Metamodellen mit semantischen Technologien

SCHLAGWÖRTER: Semantische Technologien, Ontologie, OWL 2, Constraint-Sprachen, Semantische Reasoners, Metamodellierung, Metamodellierungsplattformen, ADOxx, Metamodell Validierung, Definition von statischen Semantik

Abstract

The underlying work investigates the use of semantic technologies as a means to ensure the correctness of metamodels. In the same way that the model conforms to its metamodel, a metamodel must conform to its meta-metamodel. Conformance of metamodels is achieved by their syntactically and semantically correct definition. More specifically, a metamodel must always be in compliance with the rules enforced by syntax, and with the semantic constraints defined by semantics. While today's metamodelling tools support syntactic checks, semantic checks are not always covered. Metamodelling platforms that support semantic checks can choose among a small number of various approaches for implementation. Semantic constraints definition can either be hard-coded, or defined with the use of a constraint language, like Object Constraint Language (OCL). No matter what the approach is, definition of semantic constraints is a complex and time consuming task. Main problem is the growing number of semantic constraints within the meta-metamodel leading to the growing risk of defining contradictory checks. Throughout this work, we are going to explore the possibilities of the use of semantic technologies as a means for the definition of semantic constraints on the meta-metamodel level. More specifically, we are going to explore the possibilities of the promising ontology language Web Ontology Language 2 (OWL 2) - a self-descriptive and highly expressive language based on semantic technologies - to be used for the formal specification of semantic constraints. Additionally, we are going to explore the capabilities of using semantic reasoners for executing the semantic constraints and detecting possible violations. The applicability of this approach will be based on the meta-metamodel of the ADOxx metamodelling platform. Main objectives of this work are to achieve an easier, more declarative and easy to maintain definition of semantic constraints within the meta-metamodel, and thus a more effective and more accurate validation of metamodels.

SUBJECT AREA: Validation of metamodels using Semantic Technologies

KEYWORDS: Semantic Technologies, Ontology, OWL 2, Constraint Languages, Semantic Reasoners, Metamodelling, Metamodelling Platforms, ADOxx, Metamodel Validation, Static Semantics Definition

1 Introduction

An elaborate conceptual foundation is the main requirement of future enterprise systems. Main goal is to promote a tight mutual alignment between information systems and business. Thus, a growing interest has been shown into modelling methods – either standard or individual ones - that satisfy the domain requirements and comply with the conceptual foundations. Metamodelling platforms are software environments that allow the definition, usage and maintenance of such modelling methods, and of their main elements: (a) metamodels describing problem-specific modelling languages, (b) mechanisms and algorithms working on models and their corresponding metamodels, and (c) procedure models representing process descriptions how to apply the metamodels and the corresponding mechanisms. Some of their functional and non-functional requirements are multi-product ability, web-enablement, multi-client ability, adaptability, and scalability (Karagiannis and Kühn, Metamodelling Platforms 2002).

Metamodelling approaches have been an active research field the past twenty years, and since then they have found serious application areas in the software and information technology industries. Some primary examples are the Enterprise Model Integration (EMI) (Kühn, Bayer, et al. 2003) in the context of Enterprise Application Integration (EAI) (Linthicum 2000) for integrating metamodels of modelling languages describing different aspects of a company, Model Integrated Computing (MIC) (Ledeczi, et al. 2001), domain specific modelling languages such as the Unified Modelling Language (UML) (Object Management Group, Unified Modeling Language Specification, OMG Specifications 2003) based on the Meta Object Facility (MOF) (Object Management Group, Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) 2011), and model-driven development approaches such as Model Driven Architecture (MDA) (Object Management Group, MDA Specifications n.d.). Additionally, metamodelling approaches serve as valuable base technology to merge different modelling approaches into a domain specific modelling language, e.g. integrating UML with simulation-oriented modelling languages (Kühn and Murzek, Interoperability Issues in Metamodelling Platform 2006).

The widespread industrial and research usage of metamodelling technology can be proved by the widespread use of metamodelling platforms for specifying and implementing "domain-specific" modelling tools. Primary examples are the ADOxx metamodelling platform (BOC Group n.d.), MetaEdit+ (Metacase n.d.), Obeo Designer (Obeo Designer n.d.), GME (Institute for Software Integrated Systems n.d.) and ConceptBase (ConceptBase cc n.d.).

Today's metamodelling platforms support the definition of domain modelling languages and their concrete syntax. The definition of semantic constraints though, still remains a complex and time consuming task. Semantic constraints express the well-formedness rules applied on models. Primary examples of such constraints are the global uniqueness of properties like the object naming, the restriction of property values, the restriction of object occurrence, etc. There have been attempts to formalise the way of expressing these semantics constraints with

the use of constraint languages, with the Object Constraint Language (OCL) proposed by Object Management Group (OMG), being the most exemplary one.

By shifting our focus a level higher, it turns out that the definition of semantic constraints on the meta-metamodel level for ensuring well-formedness of metamodels remains a neglected field, which tends to be overlooked. ADOxx metamodelling platform provides full support of defining semantic constraints upon model. Exceptionally, ADOxx metamodelling platform stands out among the other metamodelling platforms for also providing adequate support for defining semantic constraints upon the meta-metamodel level, and semantic checks for their validation.

In the 2000s, domain modelling based on ontologies with Description Logics (DLs) started becoming popular (Baader, Horrocks and Sattler, Description Logics 2009). Ontologies main purpose is the representation of a specific domain and the evaluation of existing constraints over this domain for proving its consistency (Staab and Studer 2009). Web Ontology Language 2 (OWL 2) is the ontology language used and proposed by World Wide Web Consortium (W3C) for developing ontologies (Antoniou and Hamerlen, Web Ontology Language: OWL 2009). Shortly after OWL 2 having been proposed, a number of reasoning systems have been developed providing services for checking ontology's consistency or for inferring implicit knowledge.

The popularity of semantic technologies together with the widespread use of metamodelling platforms has triggered the realisation of researches with the view of integrating ontologies within the metamodelling platforms. This underlying work is an attempt to bridge the gap between metamodelling platforms and semantic technologies, by using the latter as a means to represent and ensure correctness of metamodels. Main objective of this work is to provide answers to the following questions:

- Is it possible to map a given concrete meta-metamodel and its domain specific metamodels to an ontology?
- Is expressive power of OWL 2 adequate for defining semantic constraints on the level of meta-metamodel?
- Can reasoning services be exploited for developing a validation mechanism of ontologies, and thus ensuring correctness of metamodels?

For this purpose, a metamodelling platform is needed with a concrete meta-metamodel, with a well-defined set of semantic constraints, and with a validation mechanism for detecting their violation. ADOxx metamodelling platform gathers all these features and thus will be used for the realisation of this work.

The number of research studies trying to answer to these questions and bridge the gap between metamodelling platforms and ontologies is limited. Still, they succeed to provide incontrovertible evidence of the plausibility of such integration. Within this underlying work, based always on the knowledge and the theoretical framework that previous studies provide us with, we are going to try to get one step further. Main goal is not only to stick to the theoretical proof of the realisation of such integration, but also to provide proof about the

applicability, or not, of our approach in praxis. This will be realised by transforming a concrete meta-metamodel – that of ADOxx metamodelling platform - into an ontology. This transformation concerns the meta-metamodel main concepts, the set of semantic constraints that accompanies the meta-metamodel, and the implementation of a validation mechanism for the semantic constraints using concrete ADOxx metamodels.

The Figure 1-1 depicts the focus of this work based on the logical architecture of an ontology aware metamodelling platform proposed by Zivkovic, Kühn and Murzek (2008), which extends the generic metamodelling platform architecture proposed by Karagiannis and Kühn (2002).

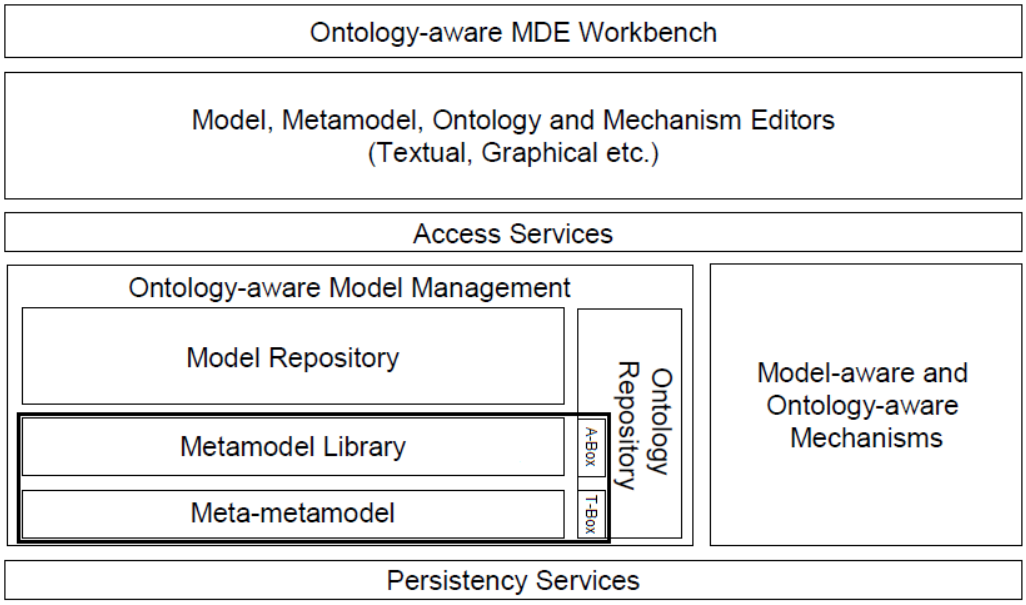


Figure 1-1: Ontology-aware metamodelling platforms logical architecture (Zivkovic, Murzek and Kühn, Bringing Ontology Awareness into Model Driven Engineering Platforms 2008)

By the end of this work, we should have a clear view of the possibilities of semantic technologies and OWL 2 to contribute to the definition of semantic constraints at the meta-metamodel level, for checking and ensuring correctness of metamodels.

The overall structure of the study takes the form of seven chapters, including this introductory chapter.

The second chapter provides an overview of the theoretical framework that the reader needs to be familiar with, in order to proceed to the following sections. This chapter is divided into three subsections, every one of which covers a scientific field studied or used within this work. The first section provides all the information needed in the field of metamodelling and of metamodelling platforms. The second section deals with all the main concepts of semantic technologies that are used within this work and aims at familiarizing the reader with terms like “OWL 2”, “Description Logics”, “Ontology”, and “Semantic reasoners”. The last section provides to the reader an overview of the related work that has been done in the field of metamodelling platforms and semantic technologies integration.

The third chapter sets the basis of our research, and it deals with the definition of an approach for mapping the ADOxx meta-metamodel and its metamodels to an ontology. For this purpose, the ADOxx meta-metamodel will be examined, analysed, and finally decomposed into smaller core elements, for achieving a concrete mapping approach, by avoiding misconceptions or a vague conceptualisation.

The fourth chapter deals with the second main task of this research. We examine the plausibility of the use of the OWL 2 for defining the ADOxx semantic constraints that accompany the meta-metamodel, within the ontology. For this purpose, the reasoning services that current semantic reasoners provide are taken into consideration. The analysis and classification of ADOxx semantic constraints into categories, is also presented, followed by the presentation of the constraint definition patterns that have been defined for reoccurring ADOxx semantic constraints.

Chapter five is concerned with the methodology used for developing a validation mechanism capable of detecting existing metamodel violations regarding ADOxx semantic constraints. Within this chapter, the design, the requirements, the architecture, the development and the functionality of the validation mechanism developed are going to be presented and analysed.

The sixth chapter deals with the performance evaluation of the developed validation mechanism. For this purpose, the performance of three widely used semantic reasoners is going to be evaluated: TrOWL (Thomas, Pan and Ren 2010), Pellet (Sirin, et al. 2007) and HermiT (Shearer, Motik and Horrocks 2008). Factors that have been found to influence in a great extend reasoning performance are going to be presented and analysed.

Finally, the conclusion gives a brief summary and review of the results, regarding the mapping approach defined, the definition of semantic constraints with the use of OWL 2, the development of the validation mechanism, and finally, of the applicability of our study. Furthermore, implications and drawbacks are summarized, triggering further future research in specific areas.

2 Background

Within this chapter, we are going to explain all the necessary concepts used within this underlying work for addressing and fully understanding the content and its main objectives. This chapter is divided into two main sections.

The first main section focuses on the area of metamodelling and on all the basic concepts that are either part of it, or arise from it. The first subsection describes what a modelling method is, as well as its main components. The second subsection focuses on the modelling hierarchy and on how the different hierarchical levels are interconnected with each other. The third section describes extensively the ADOxx metamodelling platform and its basic concepts. The last subsection briefly describes the state-of-the-art, as far as semantic constraint definition is concerned.

The second main section of this chapter focuses on semantic technologies. More specifically, the first subsection describes in detail the concept of ontology and its main building components. The second subsection refers to the OWL 2 ontology language and to its expressive power. The third subsection focuses on DLs, as it is considered to be the basis of the OWL 2 ontology language. Additionally, main basic concepts of OWL 2 like the Open World Assumption (OWA), the Unique Name Assumption (UNA) etc. are described. The fourth subsection is a brief reference to how OWL 2 expressive power can be extended with the use of rules, like for instance the Semantic Web Rule Language (SWRL). The fifth subsection focuses on the semantic reasoners and their services for inferring ontology's knowledge.

The last section of this chapter describes the related work and gives and overview of the current state of defining semantic constraints in correlation to the existing meta-metamodels, with the use of the existing constraint languages.

2.1 Metamodelling

"A model is a simplification of a system built with an intended goal in mind." (Bézivin 2001)

According to Department of Knowledge Engineering of the University of Vienna, models can be divided into two main categories: iconic and linguistic models (Karagiannis and Höfferer, Metamodels in action: An overview 2006).

Iconic models, also known as non-linguistic models, consist of signs and symbols. On the other hand, linguistic models are made up of basic primitives, like for instance signs, characters and numbers that do not usually contain an apparent relationship to the part of reality being modelled.

The latter ones can be further divided into two subcategories according to their language type. The language can be either into textual languages, or graphical/diagrammatic ones.

The different model types imply the wide variety of the purposes of their usage. Models can be used not only for visualization purposes, but also for specification and documentation purposes, in order to explain, analyse and optimize the system under study, by illuminating uncertainties, suggesting efficiencies, demonstrating trade-offs, collecting data and discovering new questions.

The next step before talking about metamodels is to clarify how models are actually built. At this point, the concept of the modelling method comes to the foreground, which describes the modelling constructs of a modelling language.

2.1.1 Modelling method

The terms "modelling method" and "modelling language" among the scientific community are used synonymously. However, a modelling method is considered to be more abstract term in comparison to a modelling language, given the fact that a modelling language is one of the necessary parts of the modelling method.

According to the framework for modelling methods proposed by Karagiannis and Kühn (2002), a modelling method is divided into the three following main components:

- modelling language
- modelling procedure
- mechanisms and algorithms

Within this present work, our focus lies on the modelling language which is described by three main parts; *notation*, *syntax* and *semantics* (Karagiannis and Kühn, Metamodelling Platforms 2002).

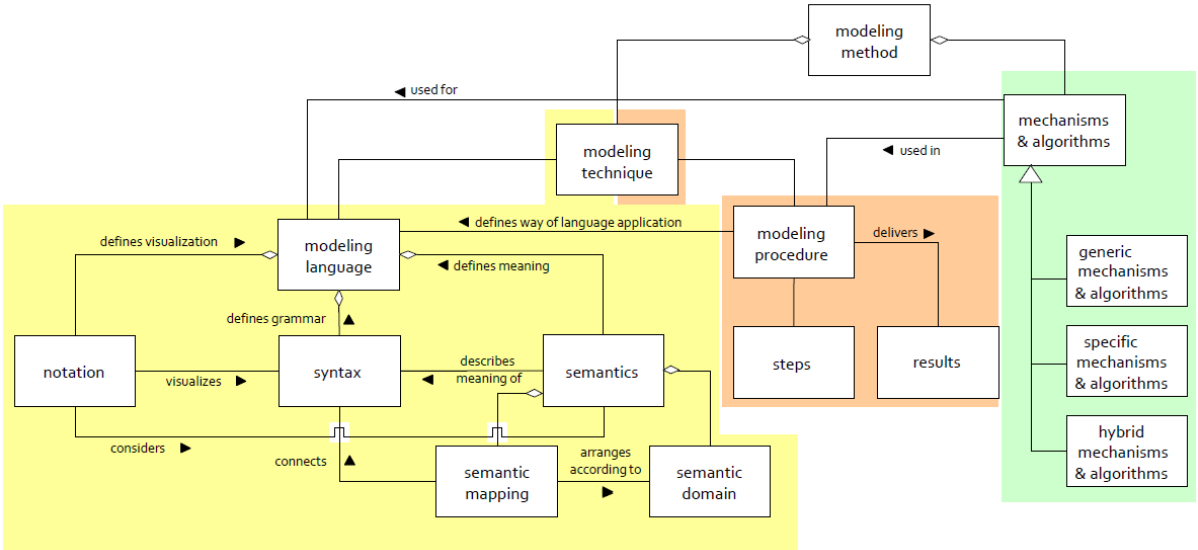


Figure 2-1: Framework for modelling methods proposed by Karagiannis and Kühn (Karagiannis and Kühn, Metamodelling Platforms 2002)

The *notation* describes the visualization of a modelling language. Notation can be differentiated into *static* and *dynamic*. *Static* notation does not take into consideration the state of the modelling constructs during the modelling process. On the other hand, *dynamic* notation considers the model state by splitting the notation in a representation and a control part. The representation part maps the static approach, and at the same time the control part defines rules which influence the representation depending on the model state.

The *syntax* describes the elements and rules for creating models. It is considered to be a specification of the modelling language constructs, of their properties and their relationships. For modelling languages, two major approaches exist to describe their syntax; graph grammars or metamodels. The basic notions of the modelling language are defined in a precise way, with structural constraints (e.g. to express containment relations, or type correctness for associations), multiplicities and implicit relationships (such as inheritance, refinement) (Kleppe 2007).

The *semantics* describes the meaning of a modelling language and consists of a semantic domain and the semantic mapping. The semantic domain may describe the meaning by using ontologies, mathematical expressions etc. Semantic mapping connects the syntactical constructs with their meaning defined in the semantic domain. Semantics can be differentiated into *dynamic* and *static* semantics.

Dynamic semantics permits representing the dynamic behaviour of a system and the operational properties.

Static semantics is considered to be well-formedness rules (WFRs). They are used to lay down complex constraints, also known as semantic constraints, which rule out illegal combinations of concepts in the modelling language (Petrascu and Chiorean, Towards Improving the Static Semantics of XCore 2010). They are usually formalized as invariants on the metamodel, using a constraint language like OCL. (Harel und Rumpe 2004). This semantics does entail neither a semantic domain, nor a semantic mapping. It further constraints the syntax.

2.1.2 Modelling hierarchy

Metamodelling is nowadays considered to be a critical part of a modelling language definition: without a precise, consistent, and validated metamodel specification, it is difficult to explain the language, build tools to support it, and produce consistent and unambiguous models (Paige, Brooke and Ostoff 2007).

The metamodel for a model is exactly what the grammar of a programming language for a programme (Bézivin 2005). Based on the four-layer metamodelling architecture (Object Management Group (OMG), Meta Object Facility (MOF) Specification 1.4 2001) as we can see in Figure 2-2, a metamodel should conform to its meta-metamodel exactly in the same way that a model conforms to its metamodel.

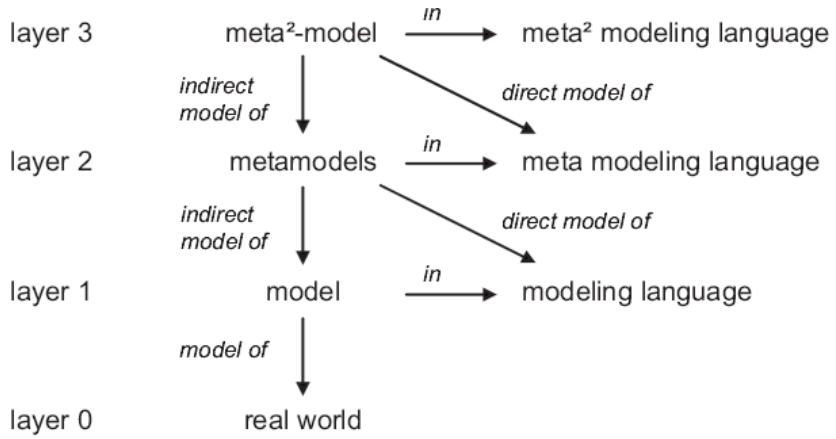


Figure 2-2: Four-layer metamodelling architecture (Karagiannis and Höfferer, Metamodels in action: An overview 2006)

Given the four-layer metamodelling architecture, the meta-metamodel defines the constructs available for the definition of modelling languages. Common metamodelling constructs are concepts like “class”, “relation”, “attribute”, “model type” etc. Currently there is a number of available implemented meta-metamodels for different modelling frameworks. The MOF (Object Management Group (OMG), Meta Object Facility (MOF) Specification 1.4 2001) is a standard meta-metamodel used to build the OMG-based family modelling languages like UML. Another implementation of the meta-metamodel is the Ecore (Steinberg, et al. 2009), the meta-metamodel of the Eclipse Modelling Framework (EMF). Comparably, the ADOxx metamodelling platform implements the ADOxx meta-metamodel, which is optimized for the rapid definition of the visual modelling languages for enterprise modelling (Zivkovic, Kühn and Murzek, An Architecture of Ontology-aware Metamodelling Platforms for Advanced Enterprise Repositories 2009).

Metamodelling platforms are software environments allowing the definition, usage and maintenance of a method elements and metamodels describing problem-specific modelling languages. Within the next section, the ADOxx metamodelling platform and its meta-metamodel are going to be extensively described.

2.1.3 ADOxx metamodelling platform

Developing modelling tools based on the notion of a Domain Specific Modelling Language (DSML) requires the use of a development environment which provides the needed means and capabilities (Karagiannis, Fill, et al. 2012).

ADOxx metamodelling platform is a metamodelling-based development and configuration environment to create domain-specific modelling products, developed by BOC group. Taking into account that ADOxx metamodelling platform is DSM oriented, defining a modelling language for a product requires that the platform, the product domain and the specific rules of the product are taken into account.

ADOxx metamodelling platform gathers the majority of important features that a metamodelling platform should have (Karagiannis and Kühn, Metamodelling Platforms 2002):

- It is an extensible, repository-based metamodelling platform
- It can be customized using metamodelling techniques
- Platform kernel provides basic modules for managing models and metamodels
- It is realized on a component-based, distributable, and scalable architecture
- The meta-metamodel, most important element of the platform, defines all the necessary concepts

An important element of ADOxx metamodelling platform is the meta-metamodel. The meta-metamodel defines the general concepts available for method definition and method usage such as "metamodel", "model type", "class", "relation", "attribute" etc.

The picture below (Figure 2-3) is an excerpt of the ADOxx meta-metamodel and its conceptual view.

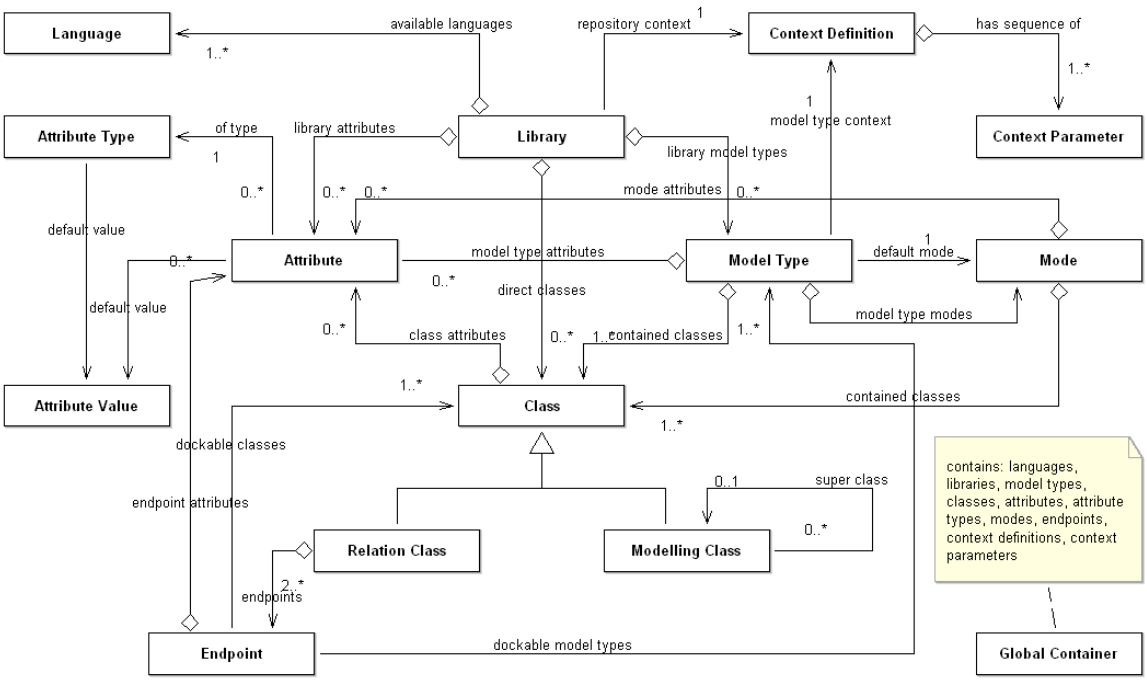


Figure 2-3: Excerpt of ADOxx meta-metamodel

According to the figure above, a *Library* is considered to be the container of all model types and transitively of all metamodel elements needed for a metamodel. A library can have one or more than one *languages*. An *attribute* is considered to be a property assigned to an element, which assigns to it specific behavioural characteristics. Attributes can be assigned to most of ADOxx meta-metamodel elements (including a library). Attributes cannot be assigned to themselves. An attribute is of a defined *attribute type* (e.g. integer, boolean, string), which defines the set of values that may take, and usually carries an *attribute value* (e.g. in case of a Boolean attribute the value would be either “true”, or “false”).

A *model type* is considered to be a container of classes and relations. In order for a library to be valid, it must contain at least one model type. A *class* represents a description of a particular modelling object. A class can be differentiated into a core class and a relation class. A *relation class* describes the relationship between two or more classes, or model types, and in contrast to core classes, they do not support inheritance. A relation class has two *endpoints* which define the classes/ model types that the relation can connect.

A model type can finally have one or more *modes*, which specify which classes are visible and possible to be used in which mode. *Context parameters* finally define a particular context and hold the list of possible static (e.g. Language: en, de etc.) or dynamic (e.g. Version: 1…-…n) values, and are contained as an ordered collection to a *context definition*.

ADOxx metamodelling platform is tightly coupled with a well-defined set of static semantics. Well-formedness of metamodels is ensured by checking their compliance to the static semantics, expressed as semantic constraints, with the corresponding semantic checks. Within ADOxx metamodelling platform, these semantic checks are also known as library checks. Currently, ADOxx metamodelling platform has two hundred seventeen (217) semantic checks for semantic constraints, hard-coded in C++.

The ADOxx meta-metamodel provides a reach set of customization options for metamodels. This leads to a high language expressive power, but at the same time increases the complexity. In the following sections, a selected set of these options will be introduced, without getting into further detail regarding the whole semantics and mechanisms behind it, in order to illustrate the complexity of the metamodelling rules.

2.1.3.1 ADOxx library

A library has a unique language independent name, i.e. library's unique identifier. Additionally, a library has a context definition, and a number of interface texts that equal to the number of the languages assigned to the library.

Library properties (Figure 2-4) define whether a library will have a repository or not, and in case it has if it is going to be with a time filter. The time filter can either be with time periods, or not. A library has a repository by default, where all models and their objects are stored.

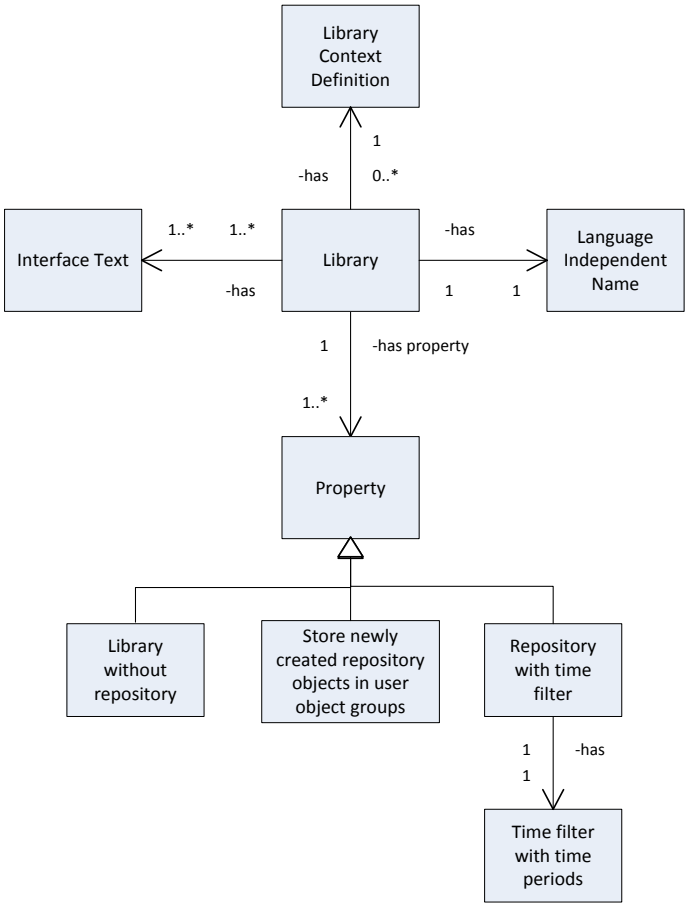


Figure 2-4: Library - ADOxx customization properties

The Figure 2-4 reveals that not all properties can be applied simultaneously. The Table 2-1 gathers all the enforced/forbidden by the platform library property combinations.

Table 2-1: ADOxx library - enforced property combinations

Enforced property combinations	**Library without repository**	**Store newly created repository objects in user object groups**	**Repository with time filter**	**Time filter with time periods**
1	True	-	-	-
2	False		False	-

2.1.3.2 ADOxx model type

Definition of a model type requires a unique language independent name and a context definition. Additionally, the number of the languages assigned to the library defines the number of the interface texts that may be filled in.

Model type properties (Figure 2-5) define:

- whether it will be visible in the modelling editor
- whether the result of an analysis query can be saved as a model of this type
- whether it will be available for modelling purposes for the end user

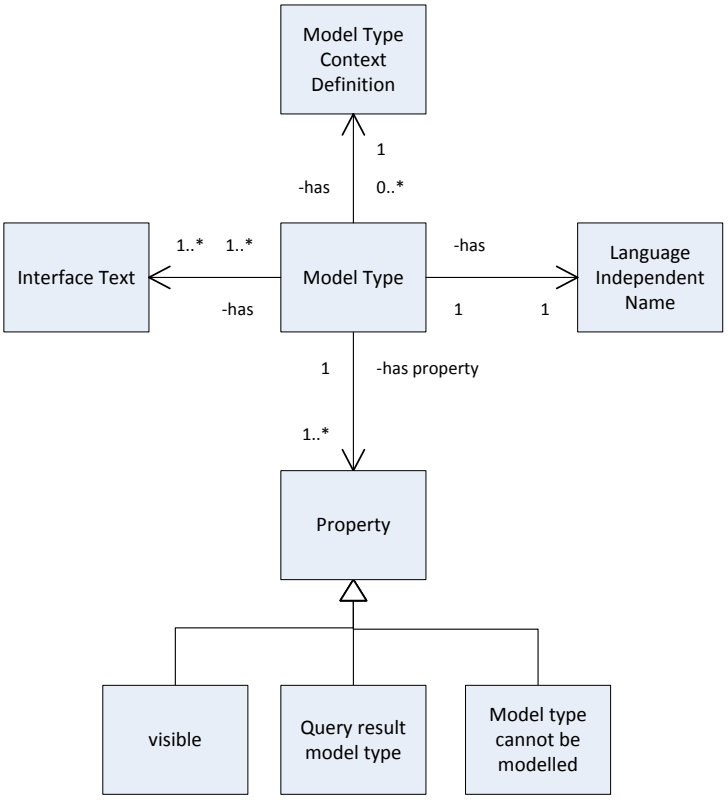


Figure 2-5: Model type - ADOxx customization properties

In the case of model type properties, there is no enforced or forbidden by the platform property combination.

2.1.3.3 ADOxx modelling class

A complete modelling class definition includes, apart from the unique language independent name and the interface texts, the definition of its super class, so that the class hierarchy can be defined.

Class properties (Figure 2-6) define:

- whether the class is visible for modelling purposes
- whether the class is a repository, or a modelling one
- whether the class can be instantiated or not (abstract)
- whether it is time filter relevant (in case that it is a repository class)

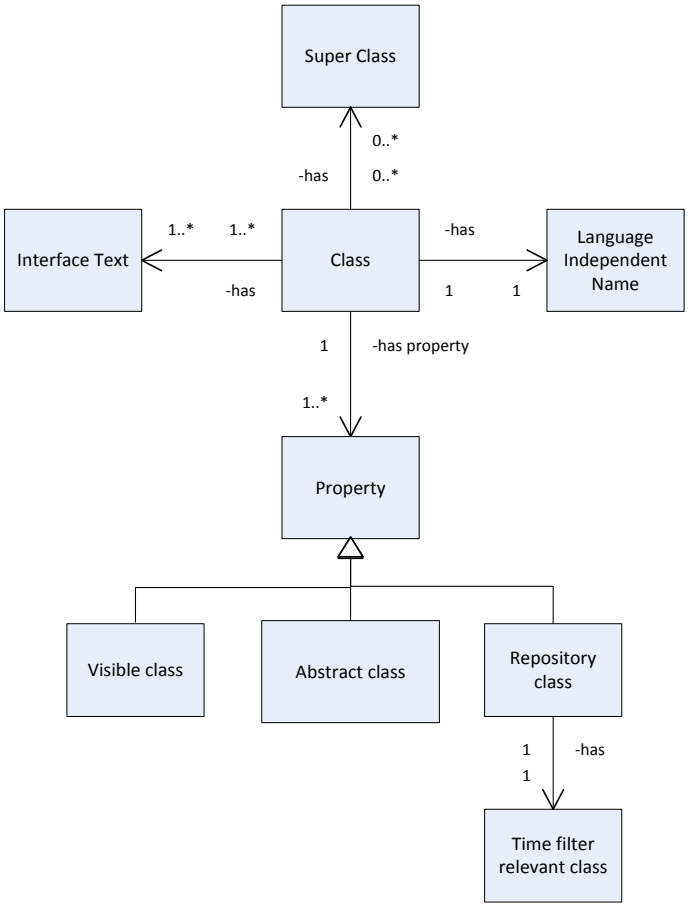


Figure 2-6: Modelling class - ADOxx customization properties

The main differences between a modelling and a repository class can be summarized within the following:

- A *repository* class has the following characteristics:
  - It is managed in the object repository i.e. in the object catalogue
  - It can be reused out of the object repository in one or more models
  - Its attributes hold the same attribute values in all occurrences
- A *modelling* class, in contrast to a repository one, has the following characteristics:
  - It is created and managed directly in one single model
  - It cannot be reused in other models
  - Its attributes are always model-context specific

The enforced/forbidden property combinations of modelling class properties are gathered within the Table 2-2.

Table 2-2: ADOxx modelling class - enforced property combinations

Enforced property combinations	Visible	Abstract	Repository class	Time filter relevant
1			False	-
2	-	True		-

2.1.3.4 ADOxx relation class

Relation class definition, respectively to modelling class, requires a unique language independent name as an identifier. In addition, interface texts can be defined, for every language assigned to the library. Main difference from a modelling class is that it represents a relation between two modelling classes. Additionally, there is no taxonomy among relation classes.

Relation class properties (Figure 2-7) allow us to define:

- whether it is going to be visible for modelling purposes
- whether it is repository or modelling
- if it exists only in a specific model context
- whether it can connect modelling classes that belong to different model types (interref)
- if it is a sub reference, i.e. used for referring to sub-models
- whether it expresses a type of
    - reflexivity
    - composition
    - ownership

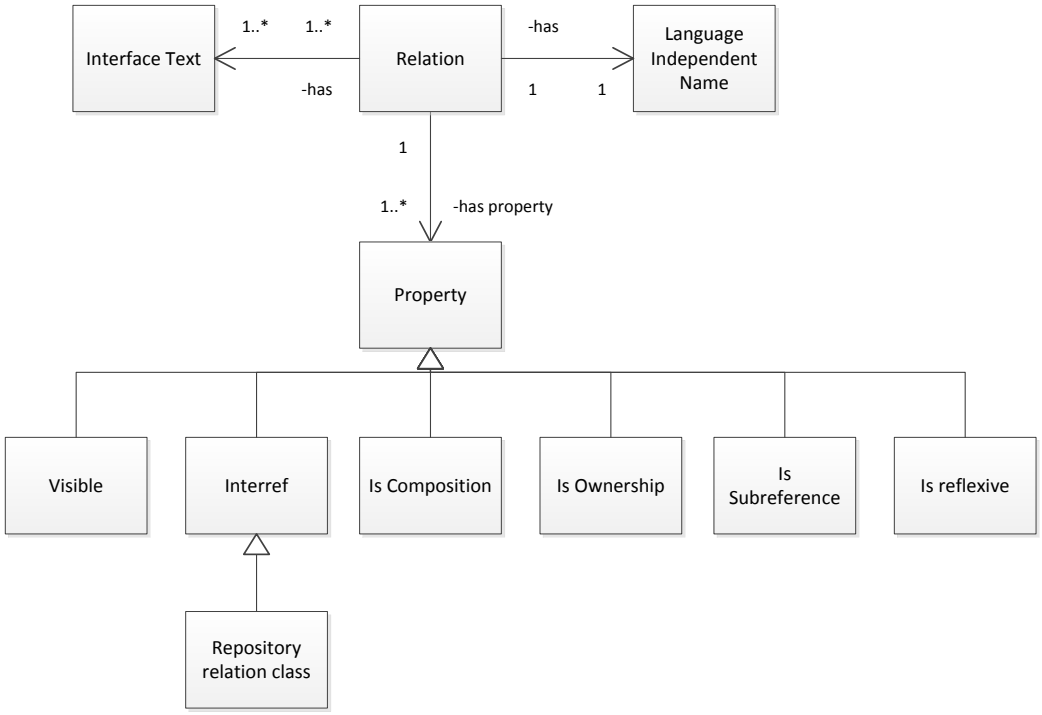


Figure 2-7: Relation class - ADOxx customization properties

The Table 2-3 gathers the enforced/forbidden property combinations that can be assigned to a relation class.

Table 2-3: ADOxx relation class - enforced property combinations

Enforced property combinations	**Visible**	**Repository relation class**	**interref**	**Composition**	**Sub-reference**	**Ownership**	**Reflexive**
1				True	-	-	
2		True	True				

2.1.3.5 ADOxx attribute

Attribute definition includes a unique language independent name, which serves as a unique identifier. It also includes the definition of interface texts, and the specification of attribute's type (i.e. string, integer, etc.). For each attribute, the default value can also be defined.

Attribute properties (Figure 2-8) allow us to specify:

- Whether it is a class attribute (provides information only for a class, and not for a class instance)
- Whether it is a model context specific attribute (its value varies among different model context definitions)
- If its value is restricted to be language independent (has the same name, regardless the language)
- Whether it is a system attribute
- Whether it is a time filter attribute

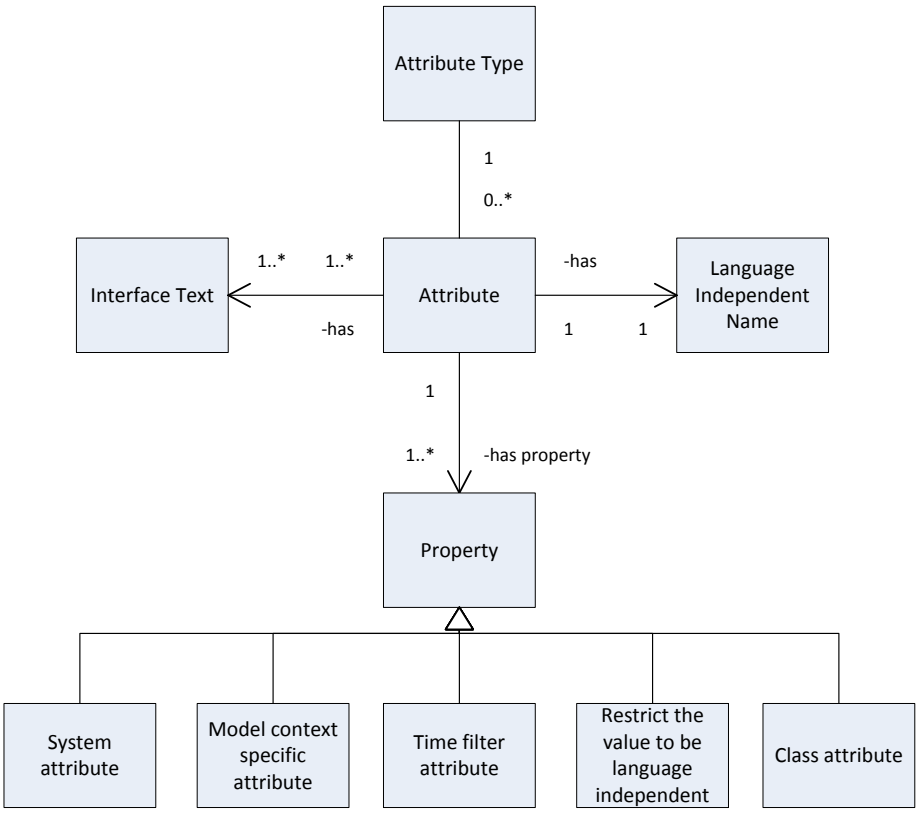


Figure 2-8: Attribute - ADOxx customization properties

The Table 2-4 gathers the enforced/forbidden by the platform attribute property combinations.

Table 2-4: ADOxx attribute - enforced property combinations

Enforced property combinations	Model context specific attribute	Time filter attribute	Restrict value to be language independent	Class attribute	Attribute Type
1	True			-	
2		-	-		Not UTC

2.1.3.6 ADOxx endpoint

Within endpoint definition, the type of the endpoint has to be defined. An endpoint can be either "From", or "To", depending on whether it refers to the source or to the target of the relation that the endpoint belongs to. Interface texts are to be defined, as well.

Endpoint properties (Figure 2-9) allow us to specify:

- whether it is visible
- whether its targets are restricted to be only modelling classes, or model types
- whether its targets are restricted to be only model types
- whether it is time filter relevant

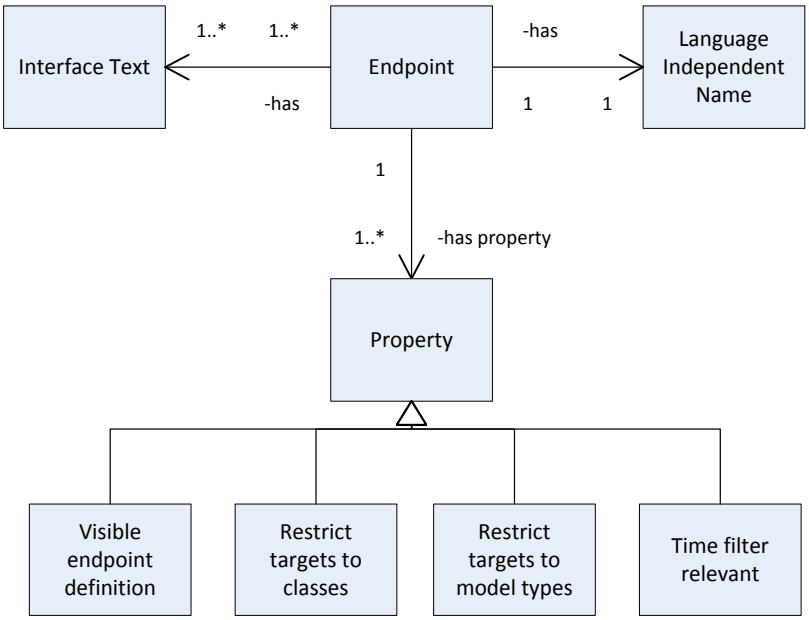


Figure 2-9: Endpoint - ADOxx customization properties

The Table 2-5 gathers the property combinations of an endpoint enforced/forbidden by the platform.

Table 2-5: ADOxx endpoint - enforced property combinations

Enforced property combinations	Visible endpoint definition	Restrict targets to objects	Restrict targets to models	Restrict targets to repository instances	Time filter relevant
1		False		-	
2		True	-		
3		-	True	-	

2.1.4 Semantic constraints definition: state-of-the-art

Based on the definition given earlier in the section 2.1.1, static semantics defined within the meta-metamodel assigns a semantic interpretation to the main concepts of a metamodel. Static semantics is expressed as semantic constraints upon the metamodel. Semantic constraints are semantic information attached to an element, and they indicate restrictions that must be enforced by correct design of a metamodel. A metamodel is considered to be well-formed when it satisfies its semantic constraints. Conformance of metamodels to the semantic constraints is considered to be a crucial task. A metamodel must always be in compliance with the defined semantic constraints.

While syntactic checks are mainly enforced by the metamodelling platform, semantic checks for ensuring compliance to the semantic constraints are not always covered and require additional effort. Metamodelling platforms that support semantic checks can choose among a small number of various approaches for implementation. In the case of metamodelling platforms like ADOxx, semantic constraints and their checks are hard-coded with a use of a programming language. In the case of other meta-metamodels like MOF, ECore (Petrascu and Chiorean, Proposal of a Set of OCL WFRs for the ECore 2009) or XCore (Petrascu and Chiorean, Towards Improving the Static Semantics of XCore 2010) semantic constraints are defined with the use of a constraint language like OCL, and validated with the use of appropriate mechanisms.

No matter what the approach, definition of semantic constraints is a complex and time consuming task. It requires the use of a language for the formal specification of semantic constraints, and an engine to execute the constraints, and detect their violations – semantic checks. Main problem is the growing number of semantic constraints within the meta-metamodel leading to the growing risk of defining contradictory ones.

ADOxx metamodelling platform supports semantic checks, for validating metamodels against the semantic constraints, making it possible to produce correct and high-quality metamodels. The extended ADOxx meta-metamodel though, in addition to the hard-coded implementation of semantic constraints makes it quite hard to maintain or update the existing set of semantic constraints.

An extended literature research has shown that the research field regarding the definition of semantic constraints on the metamodel level has been much more active than that of defining semantic constraints on the meta-metamodel level.

2.1.4.1 OCL

OCL (Object Management Group (OMG), Object Constraint Language (OCL), version 2.2 2010) is considered to be one of the most dominant constraint languages. OCL's main usage, beyond querying, is to describe semantic constraints, by specifying invariants on classes, by describing pre- and post-conditions on methods, and by specifying initial and derived rules over a specified model. It is an unambiguous formal language and a pure specification language. It is mainly applied within UML metamodel, but it has been not a long time ago though, since some research has been taking place concerning the use of OCL for semantic constraint definition upon MOF.

Considering an example from the MetaGME homepage, suppose the finite state machines in the target domain must not allow state transitions from one state to itself. A UML class diagram alone cannot specify such a rule. Thus, the following OCL expression must be attached to states:

Example 2-1: Example of a semantic constraint in OCL

```
self.transTo -> forAll(s | s <> self)
```

Self and *forAll* are OCL keywords, while *transTo* is a role name of the transition association.

2.1.4.2 Alloy

Alloy (Alloy Constraint Language n.d.) is another declarative specification language main purpose of which is to express structural constraints and behaviour in a software system. An Alloy module consists of a module header, a set of imports and zero or more paragraphs. The module header is a name of the module where signatures, constraints, assertions and commands are defined. An import allows including additional modules. Furthermore, a paragraph can either be a signature declaration, a constraint, an assertion or a command.

Constraints are defined by facts, predicates and functions. Facts are invariants; i.e, their associated constraints always hold. Predicates are named constraints, which can be used in diverse contexts. The difference between fact and a predicate is that the first one always holds while the second one only holds when invoked. Finally, functions describe named expressions, which can be also reused in diverse contexts. Alloy provides a wide range of quantifiers, logical and comparison operators for defining semantic constraints.

The widely used "Family tree" example serves as a supplementary source for understanding reasons.

Example 2-2: Example of a semantic constraint in Alloy

```
module Family tree
sig Name { }
abstract sig Person {  name: one Name,  siblings: Person,  father: lone Man,  mother: lone Woman  }
sig Man extends Person { wife: lone Woman  }
sig Woman extends Person {  husband: lone Man  }
sig Married extends Person {  }
fact {
all p : Persons | (sole (p.parents & Man)) &&
(sole (p.parents & Woman))}
```

The expression by fact expresses the following semantic constraint: "No person can have more than one father of mother".

2.1.4.3 QVT (Query/View/Transformation)

QVT (Object Management Group, Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) 2011) is a standard set of languages for model transformation defined by the MOF. The QVT specification has a hybrid declarative/imperative nature, with the declarative part being split into a two – level architecture.

2.2 Semantic Web Technology Concepts

One of the common uses for the term semantic web is to identify a set of technologies, tools and standards which form the basic building blocks of a system that could support the vision of a web imbued with meaning (Siddiqui and Deshmukh 2012) .

The Figure 2-10 represents the concrete layers that build Semantic Technology and their place among the hierarchy.

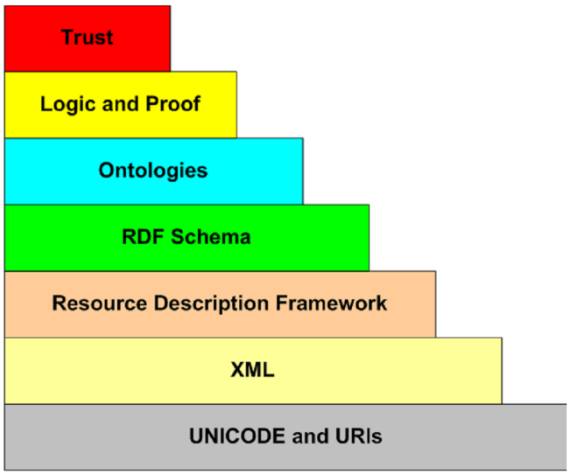


Figure 2-10: Layers of Semantic Technology (Gerber, van der Merwe and Barnard 2008)

Currently, semantic technologies are applied to a various number of industrial sectors like in healthcare, in finance or life sciences (2011 Semantic Technology Conference 2011).

Within this underlying work, our focus lies on the use of semantic technologies within the sector of metamodelling platforms and more specifically on the advantages that rise from their appliance. Among others, the reasons that encouraged the use of semantic technologies can be summarized within the following points:

- Virtualize complex relationships
- Query complex datasets
- Monitoring
- Document Processing

Among the layers of semantic technology architecture, as a brief look at the layers of semantic technology (Figure 2-10) reveals, “Ontologies” seems to be one of the main building blocks.

2.2.1 Ontology

The word “Ontology” is used among many communities, every one of which assigns to the word a domain-specific meaning. Although every domain-specific meaning is not totally irrelevant, the most radical difference can be denoted between the philosophical sense and the computational sense. In this chapter and the context of this work, our focus lies on the word “Ontology” as a term used within the computer science domain.

Computational ontologies are a means to model formally the structure of a system, i.e., the relevant entities and relations that emerge from its observation, and which are useful to our purposes (Staab and Studer 2009).

It was Gruber, in 1995 that defined “An ontology is a formal, explicit specification of shared conceptualization” (Struder, Benjamins and Fensel 1998). According to the definition above, conceptualization seems to be one of the key words while defining an ontology. Conceptualization is considered to be an abstract, simplified view of the world that we wish to represent (Gruber 1993).

Ontology structure consists of two main components: the TBox and the ABox.

TBox is an abbreviation for the Terminology Box. It is considered to be the backbone of an ontology and consists of a generalization/specialization hierarchy of concepts, i.e., a taxonomy. It could be considered as the ontology in the form of concepts and role definitions.

ABox is an abbreviation for the Assertion Box. ABox contains assertions about individuals using the terms/concepts defined in ontology’s TBox.

The Figure 2-11 is an example of the famous ”Pizza ontology” used in order to clarify the difference between TBox and ABox.

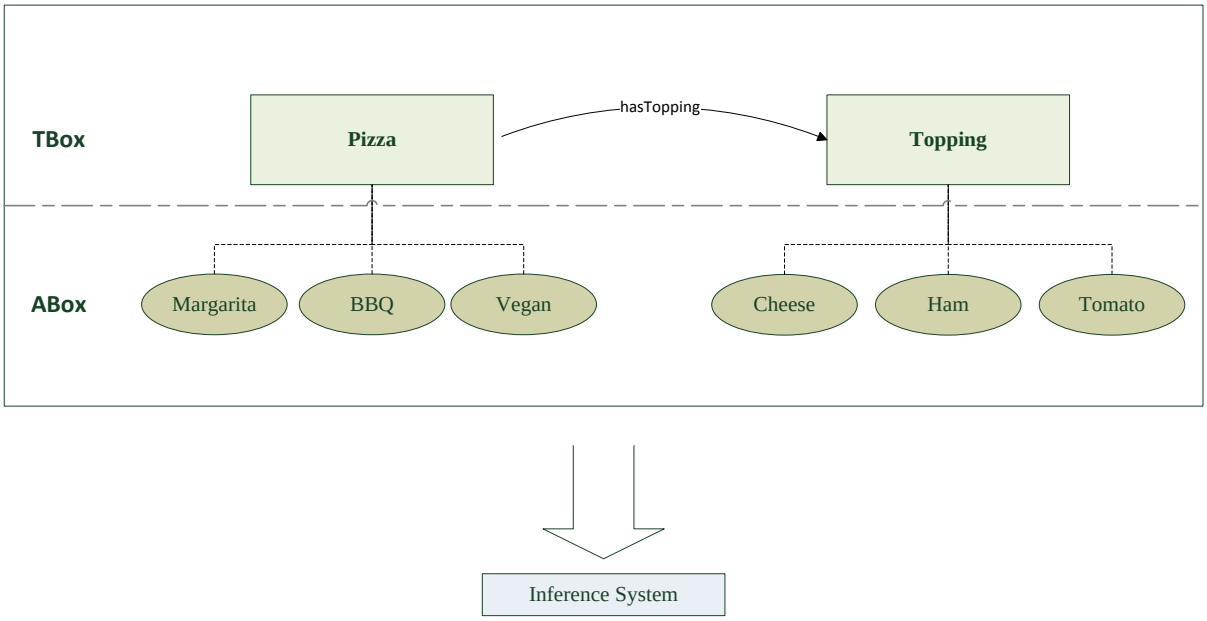


Figure 2-11: Ontology “Pizza” example - TBox and ABox

The Figure 2-12 is a graphical representation of ontology structure and its main components.

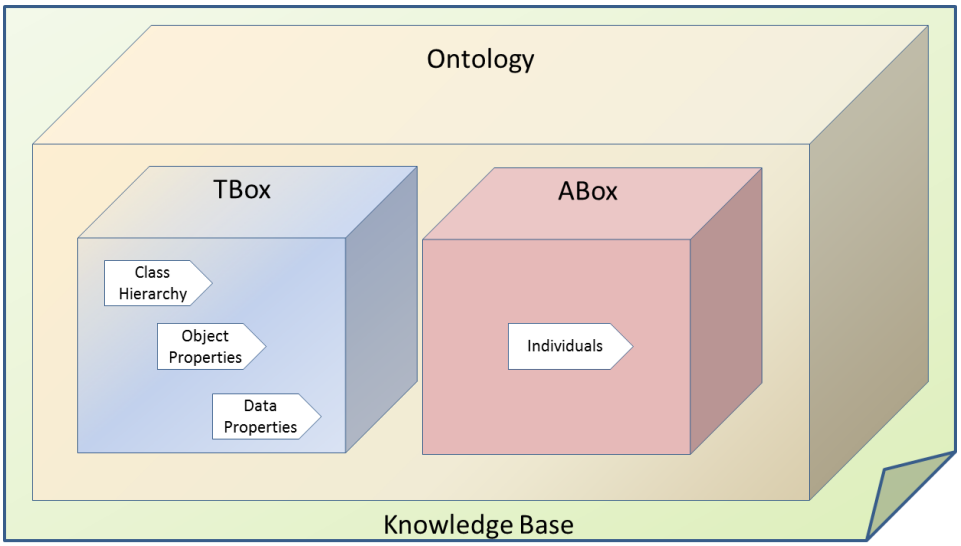


Figure 2-12: Ontology - main components

2.2.2 OWL 2

Ontologies are described by an ontology language which allows users to write explicitly formal conceptualizations of domain models. OWL 2 is proposed by W3C and is considered to be the most dominant ontology language currently.

Predecessors of OWL are languages like SHOE (Heflin, Hendler and Luke 1999) - a frame-based language with an XML (Extensible Markup Language) syntax, OIL (Ontology Interface Layer) (Fensel, et al. 2001) - the first language to combine DLs, frame languages,

and web standards, such as XML and RDF (Resource Description Framework), and DAML+OIL - which is the result of the merge of the languages DAML-ONT and OIL and they have influenced the OWL 2 in a great extent (Gómez-Pérez and Corcho 2002).

The Figure 2-13 shows the OWL predecessor ontology languages stack in the semantic web.

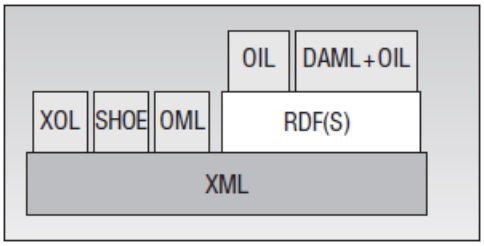


Figure 2-13: The languages stack in the semantic web (Gómez-Pérez and Corcho 2002)

Additionally, the Figure 2-14 shows the history of development of the web ontology languages.

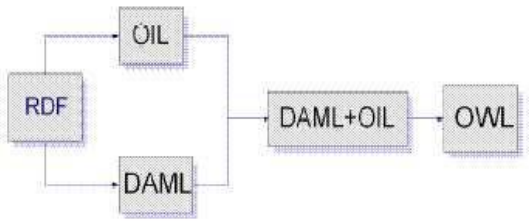


Figure 2-14: From RDF(S) schema to OWL (Ontology n.d.)

OWL 2 gathers all the requirements that an ontology language should have (Antoniou and Van Hamerlen, Web Ontology Language: OWL 2003):

- *A well-defined syntax*: importance of well-defined syntax is known from the area of programming languages, and it is a necessary prerequisite for machine-processing of information.
- *A well-defined formal semantics*: it describes precisely the meaning of knowledge and their importance is well-established in the domain of mathematical logic. Semantics is the main prerequisite for reasoning support.
- *Efficient reasoning support*: it allows checking the consistency of the ontology and the knowledge base, checking for unintended relationships and automatically classifies instances and concepts.
- *Sufficient expressive power*: complex syntax usually hides real expressive power. It refers to relational, and data type expressivity.
- *Convenience of expression*: it refers to the ability to express complex interrelations and schemas in a convenient way, in order to be human readable.

OWL 2 is about representing knowledge with machine understandable semantics, therefore, well-defined semantics is considered to be one of the most crucial characteristic of OWL 2.

Semantics allows humans to reason about the knowledge. As far as ontological reasoning is concerned, the following knowledge can be reasoned (Antoniou and Van Hamerlen, Web Ontology Language: OWL 2003):

- Class membership
- Equivalence of classes
- Consistency
- Classification

Semantics is a prerequisite for reasoning support. Knowledge embodied within an ontology can be derived manually, in case of a small-size ontology. Semantic reasoners come to simplify things and make the whole process automatic, when we have to cope with large-size ontologies, where implicit knowledge is hard to be derived.

The expressive power of OWL 2 is determined by the class and property constructors supported and by the kinds of axioms that can occur within the ontology. Of course increased expressive power inevitably leads to increased computational complexity for key reasoning problems such as entailment (Horrocks, Patel - Schneider and Van Harmelen 2003). Major reasoning task is to identify undesirable entailments like class unsatisfiability and ontology inconsistency (Horridge, Bauer, et al. 2009). OWL 2 is considered to be the most expressive and still decidable ontology language.

OWL 2 has three increasingly expressive sub-languages (Antoniou and Hamerlen, Web Ontology Language: OWL 2009): OWL Lite, OWL DL, and OWL Full.

- *OWL Full* uses all the OWL languages primitives. Main advantage of OWL Full is that it is fully compatible with RDF, both syntactically and semantically. Main disadvantage is that it has become extremely powerful to be decidable.
- *OWL DL* is a sublanguage of OWL Full, which was developed in order to gain back computational efficiency. This is achieved by restricting the way in which the constructors from OWL and RDF can be used. Main advantage is that it permits efficient reasoning support. Loss of full compatibility with RDF can be considered as the main disadvantage of this sublanguage.
- *OWL Lite* is the sublanguage that stems from a further restriction of OWL DL to a subset of the language constructors. Main advantage is that it is easier for the user to grasp and for the tool builders to implement, but its restricted expressivity remains its main drawback.

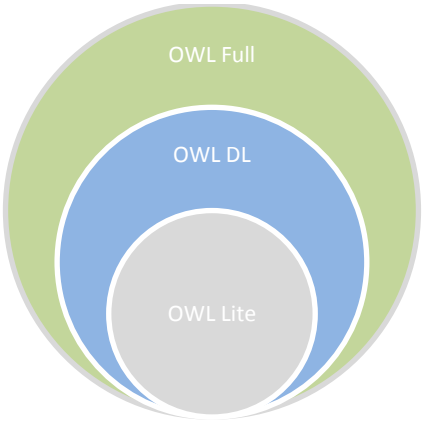


Figure 2-15: OWL 2 profiles

Apart from the three different sub-languages of OWL 2 (Figure 2-15), there is a number of different OWL 2 syntaxes. A concrete syntax is needed in order to store OWL 2 ontologies and to exchange them among tools and applications.

All syntaxes are gathered in the table below (Table 2-6).

Table 2-6: OWL 2 syntaxes (W3C)

Name of syntax	**Purpose**
RDF/XML	Interchange (can be written and read by all conformant OWL 2 software)
OWL/XML	Easier to process using XML tools
Functional Syntax	Easier to see the formal structure of ontologies
Manchester Syntax	Easier to read/write DL Ontologies
Turtle	Easier to read/write RDF triples

2.2.2.1 OWL 2 expressivity axioms

Main component of an OWL 2 ontology is considered to be a set of axioms, which are statements that declare what is true in the domain.

The Figure 2-16 represents the different sets of OWL 2 axioms.

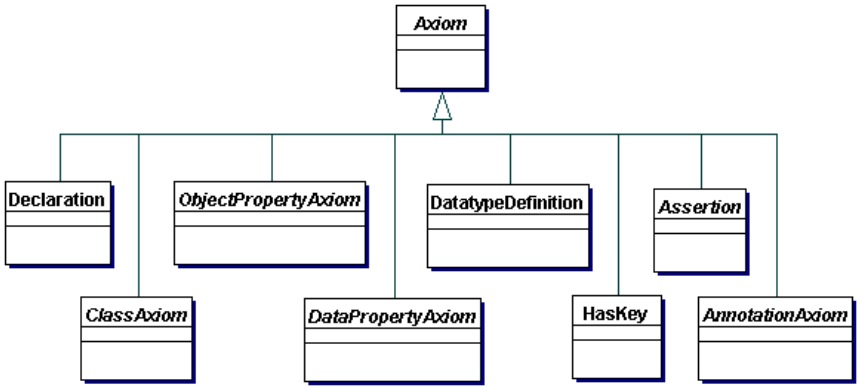


Figure 2-16: OWL 2 axioms (OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax n.d.)

Within the following sections, we are going to describe briefly three of these main sets: class, object property and data property axioms. All the used figures are taken from the W3C recommendation "OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition)" proposed in December 2012.

1.2.2.1.3 OWL 2 class axioms

OWL 2 provides axioms that allow relationships to be established between class expressions, as shown in Figure 2-17.

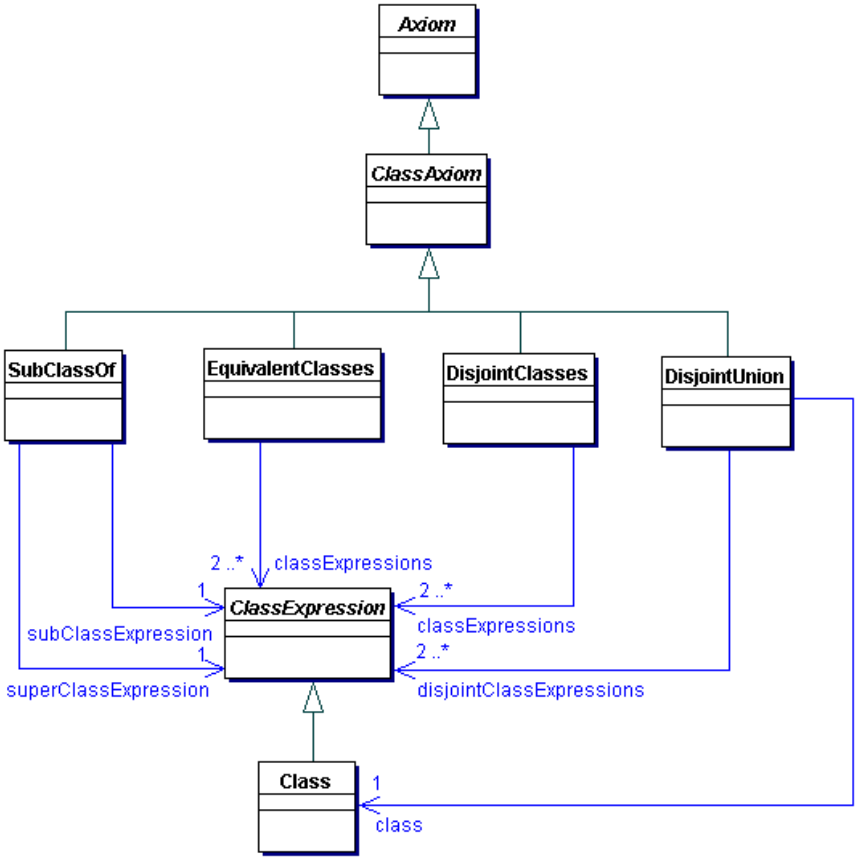


Figure 2-17: OWL 2 class axioms

The simplest form of a class axiom is a class description that states the existence of a class, using *owl:Class* with a class identifier. OWL 2 contains three language constructs for combining class descriptions into class axioms:

- *SubClassOf* allows defining that an OWL 2 class is a subset of another class.
- *EquivalentClasses* allows defining that an OWL 2 class is equivalent to another class.
- *DisjointClasses* allows defining that an OWL 2 class cannot have common members with another class.

2.2.2.1.4 OWL 2 object property axioms

Apart from the OWL 2 class axioms, there are also the OWL 2 object property axioms, which once again are used to characterize and establish relationships between object property expressions. The Figure 2-18 shows the different types of object property axioms.

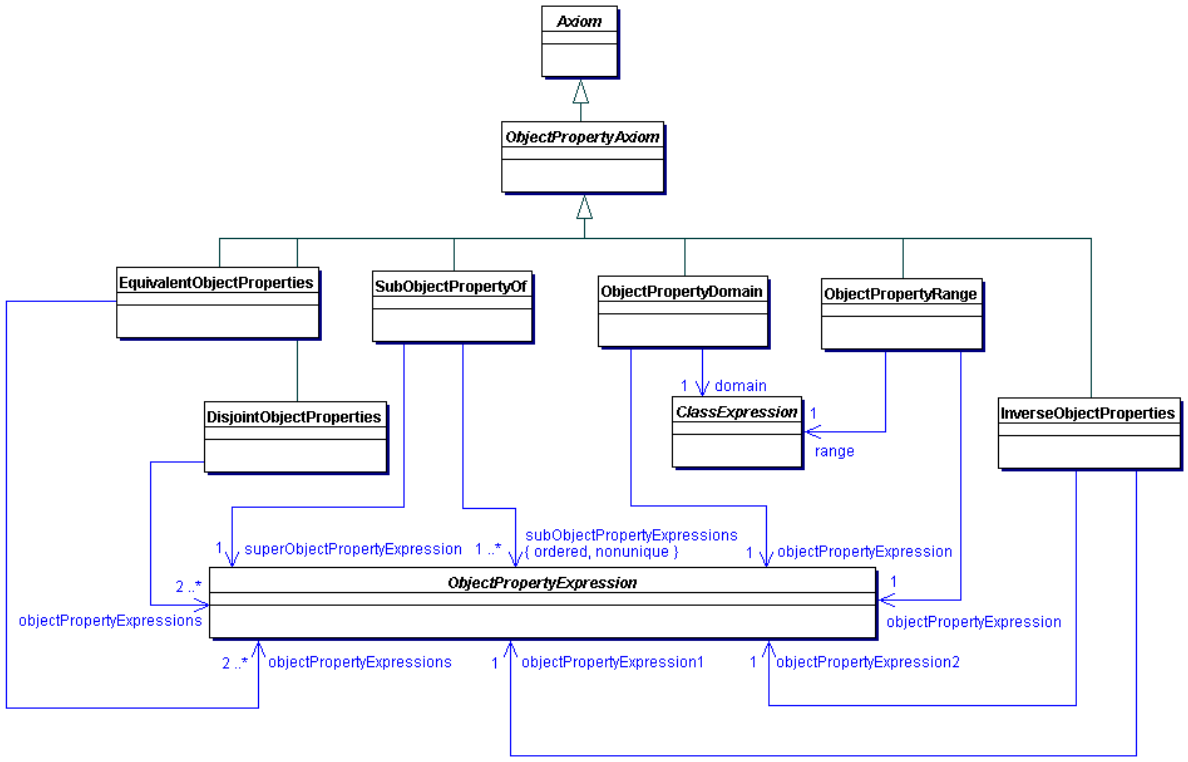


Figure 2-18: OWL 2 object property axioms

More specifically:

- *ObjectPropertyDomain* axioms are used to restrict the source individual connected by an object property to be an instance of a specified OWL 2 class.
- *ObjectPropertyRange* axioms are used to restrict the target individual connected by an object property to be an instance of a specified OWL 2 class.
- *InverseObjectProperties* axiom is used to state that two object properties are the inverse of each other.
- *DisjointObjectProperties* axiom is used to state that the extensions of several object properties are pairwise disjoint — meaning that they do not share pairs of connected individuals

Additionally, as the Figure 2-19 reveals, there are axioms which define characteristics of object properties in OWL 2.

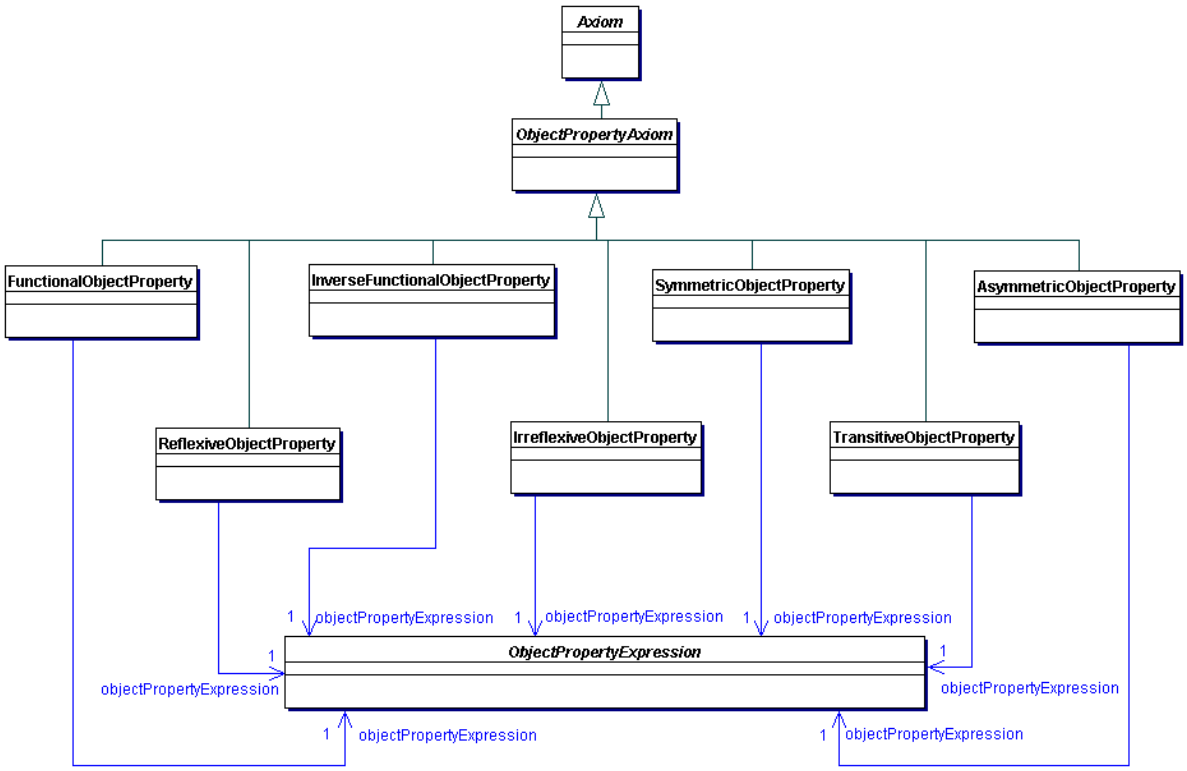


Figure 2-19: OWL 2 axioms defining object properties characteristics

More specifically:

- *FunctionalObjectProperty* axiom is used to restrict the number of target individuals connected by a property to exactly one.
- *SymmetricObjectProperty* axiom can be used to define an object property as bidirectional.
- *AsymmetricObjectProperty* defines an object property as strictly not bidirectional.
- *ReflexiveObjectProperty* axiom can be used to state explicitly that an object property can have as target and source the same individual of the same concept.
- *IrreflexiveObjectProperty* axiom can be used to state explicitly that an object property cannot have as target and source the same individual of the same concept.
- *TransitiveObjectProperty* axiom infers the connection of a starting and an ending individual with an object property, by the connection of the in-between individuals with the same object property.

2.2.2.1.3 OWL 2 data property axioms

OWL 2 provides a set of OWL 2 data property axioms. The Figure 2-20 represents the different types of OWL 2 data property axioms.

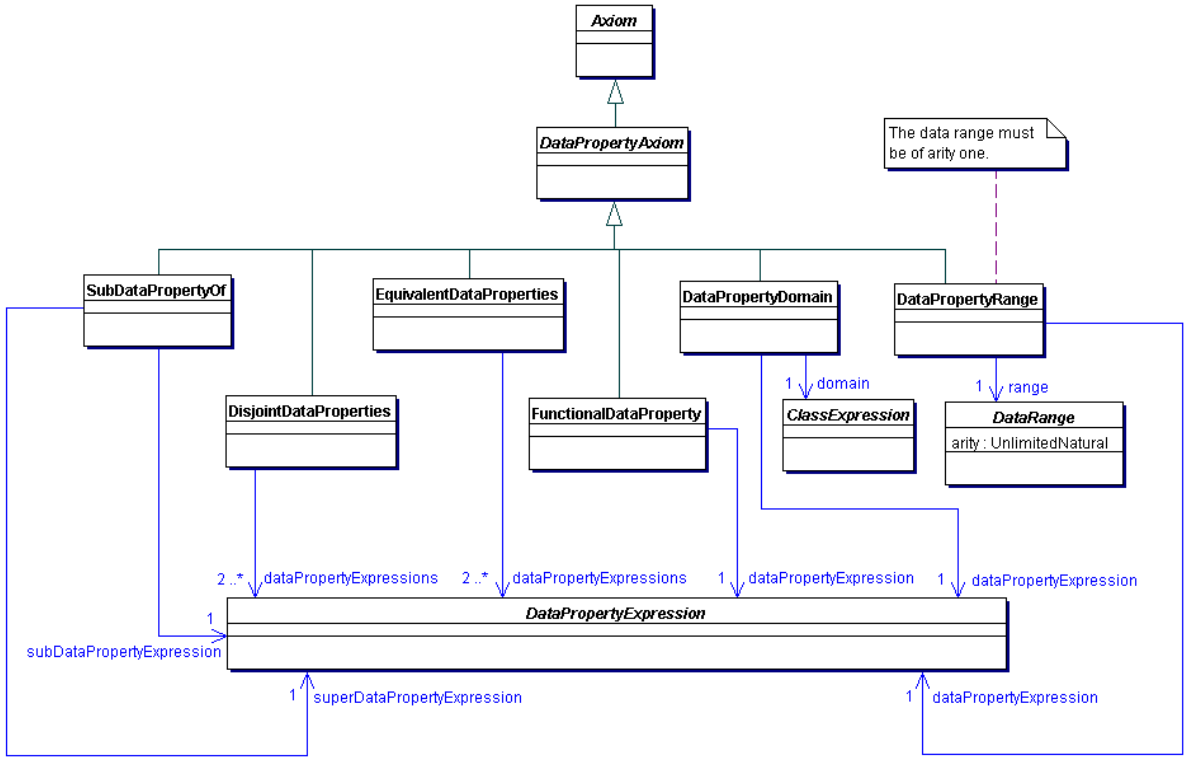


Figure 2-20: OWL 2 data property axioms

More specifically:

- *DataPropertyDomain* axiom is used to restrict individuals connected by a property to be an instance of a specified class.
- *DataPropertyRange* axiom can be used to restrict the literals pointed to by a property to be in the specified unary data range.
- *InverseObjectProperties* axiom is used to state that an object property is the inverse of another one.
- *DataPropertyRange* axiom can be used to restrict the literals pointed to by a property to be in the specified unary data range.

2.2.2.1.4 OWL 2 individual axioms

OWL 2 provides a set of OWL 2 individual axioms. Individual axioms are divided into three main categories:

- Class assertions in OWL 2
- Object property assertions in OWL 2
- Data property assertions in OWL 2

The Figure 2-21 represents only the individual assertions regarding OWL 2 classes, as these are the ones used within this underlying work.

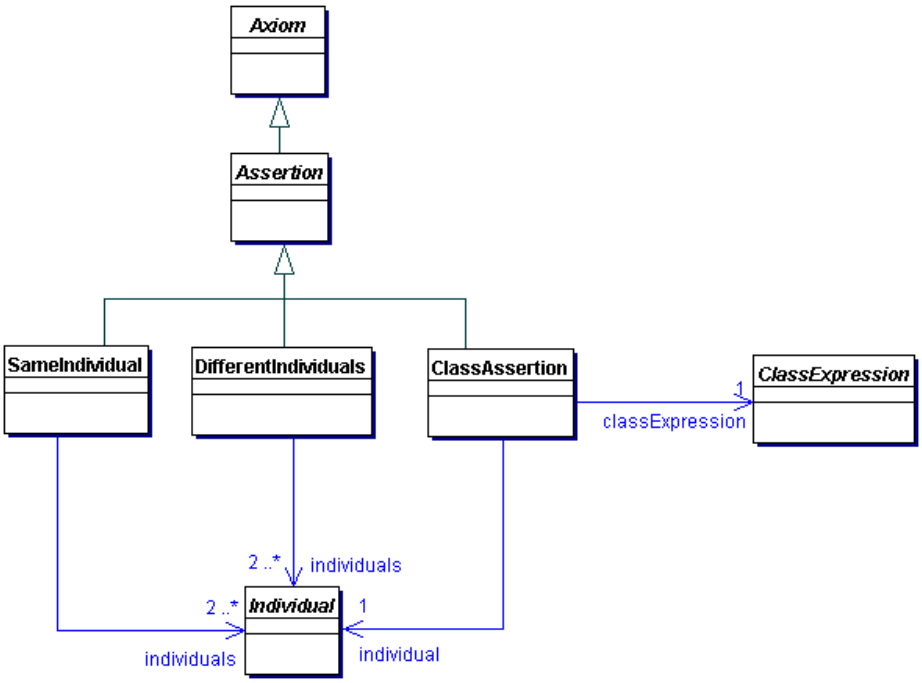


Figure 2-21: OWL 2 - class and individual assertions

The *SameIndividual* assertion states that several individuals are all equal to each other, while the *DifferentIndividuals* assertion states the opposite — that is that several individuals are all different from each other. The *ClassAssertion* axiom allows one to state that an individual is an instance of a particular class.

2.2.3 Description Logics (DLs)

Basis of OWL 2 is considered to be DLs (Baader, Calvanese, et al. 2003). DLs are a family of knowledge representation languages that can be used to represent the knowledge of an application domain in a structured and formally well-understood way. The name description logics is motivated by the fact that the important notions of the domain are described by concept descriptions, i.e. expressions that are built from atomic concepts (unary predicates) and atomic roles (binary predicates) using the concept and role constructors provided by the particular DL (Baader, Horrocks and Sattler, Description Logics 2009).

The Figure 2-22 represents the architecture of a knowledge representation system based on DLs.

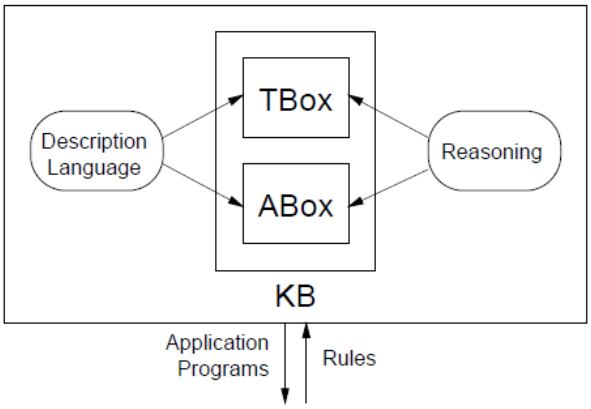


Figure 2-22: Architecture of a DLs-based knowledge representation system (Baader and Nutt, Basic description logics 2003)

A DL knowledge base consists of a set of terminological axioms and a set of assertional axioms. Basic symbols that compose the syntax of DLs are gathered in Table 2-7 accompanied by a short description and an example.

Let “A” and “C” be concepts, “a” and “b” individuals, and “R” a role.

Table 2-7: DLs syntax

Name	Concrete Syntax	Abstract Syntax
Top	TOP	$\top$
Bottom	BOTTOM	$\bot$
Intersection	A and C	$A \sqcap C$
Union	A or C	$A \sqcup C$
Negation	not C	$\neg C$
Universal restriction	all R C	$\forall R.C$
Existential restriction	some R C	$\exists R.C$
Concept inclusion	all C are A	$C \sqsubseteq A$
Concept equivalence	A is equivalent to C	$A \equiv C$
Concept definition	A is defined to be equal to C	$A \doteq C$
Concept assertion	a is a C	$a : C$
Role assertion	a is R-related to b	$(a,b) : R$

$\mathcal{AL}$ (AL - Attributive Language) was first introduced by Schmidt - Schauss and Smolka in 1991, as a minimal language that is of practical interest and as a basic description language. By extending language expressive power, we get a number of languages which all belong to the $\mathcal{AL}$ - languages family. Apart from this language there is the $\mathcal{FL}$ (FL - Frame based description Language), which is obtained by $\mathcal{AL}$ by disallowing atomic negation. In contrast to the previous languages, whose common core is concept constructors intersection and value restriction, this language allows the intersection of concepts and existential quantification (but not value restriction).

By adding one, of the following extensions to these three main DLs languages, their expressive power is enriched and extended (Table 2-8).

Table 2-8: DLs extensions and meaning

Symbol	Meaning
$\mathcal{F}$	Functional properties
$\mathcal{E}$	Full existential quantification
$\mathcal{U}$	Concept union
$\mathcal{C}$	Complex concept negation
$\mathcal{H}$	Role hierarchy
$\mathcal{R}$	Limited complex role inclusion axioms
$\mathcal{O}$	Nominals
$\mathcal{I}$	Inverse properties
$\mathcal{N}$	Cardinality restrictions
$\mathcal{Q}$	Qualified cardinality restrictions
$(\mathcal{D})$	Use of data properties

The symbol $\mathcal{S}$ is an abbreviation for $\mathcal{ALC}$ with transitive roles (also known as $\mathcal{ALC}_{R^+}$). Prominent members of the $\mathcal{S}$ - family are:

- $\mathcal{SIN}$, which extends $\mathcal{S}$ with number restrictions and inverse roles

- $\mathcal{SHIF}$, which extends $\mathcal{S}$ with role hierarchies, inverse roles and number restrictions
- $\mathcal{SHIQ}$, which extends $\mathcal{S}$ with role hierarchies, inverse roles and qualified number restrictions

The suitability of DLs as ontology languages has been highlighted by their role as the foundation for several web ontology languages, including OWL 2. More specifically, OWL 2 is based on DLs $\mathcal{SROIQ}^{(\mathcal{D})}$.

$\mathcal{SROIQ}^{(\mathcal{D})}$ is considered to be the descendent of $\mathcal{SHOIN}^{(\mathcal{D})}$ underlying OWL DL. Although $\mathcal{SHOIN}^{(\mathcal{D})}$ could be said that it provided a great amount of expressive means, it turned out that it lacked some that could be easily added without causing too much trouble for semantic automated reasoning. The Table 2-9 clarifies for what the name $\mathcal{SROIQ}^{(\mathcal{D})}$ stands for:

Table 2-9: $\mathcal{SROIQ}^{(\mathcal{D})}$ expressive power

Symbol	Name
$\mathcal{S}$	An abbreviation of $\mathcal{ALC}$ which stands for Attribute Language with transitive properties
$\mathcal{R}$	Limited complex role inclusion axioms; reflexivity and irreflexivity; role disjointness.
$\mathcal{O}$	Nominals
$\mathcal{I}$	Inverse properties
$\mathcal{Q}$	Qualified property restrictions
$(\mathcal{D})$	Use of Data Properties

2.2.3.1 OWL 2 and Internationalized Resource Identifier (IRI)

In OWL 2, all names are global. Therefore, every name, no matter if it is an OWL 2 class name, an OWL 2 object property name, or an individual name, has to be unique. In OWL 2 this is called IRI, and it is globally unique.

2.2.3.2 OWL 2 and the Open World Assumption (OWA)

OWL 2 is based on the OWA, which is a property inherited by the First Order Logic (FOL). Fundamental principle of the OWA is that *everything is true, unless it can be proven false.*

On the contrary, Closed World Assumption **(**CWA), which is the main competitor against OWA, states that *everything that cannot be found to be true is automatically assumed to be false.*

The Table 2-10 summarizes the main differences between the OWA and the CWA.

Table 2-10: OWA Vs. CWA

Closed World Assumption (CWA)	**Open World Assumption (OWA)**
Negation as Failure (NaF)	Negation as Contradiction (NaC)
Anything that cannot be found is considered to be false	Anything that cannot be found might be true, unless it can be proven false
Reasoning: any world that has to do with databases	Reasoning: any world consistent with Ontologies, DLs
CWA implies that *everything we don't know is false*	OWA states that *everything we don't know is undefined*

Keeping in mind that knowledge is unlimited, it becomes quite clear why OWL 2 is based on the OWA and not on the CWA, although the latter one is quite frequently needed for representing knowledge of software systems making use of a Database Management System.

2.2.3.3 OWL 2 and Unique Name Assumption (UNA)

UNA states that two elements with different names are considered to be different. OWL 2 does not follow this assumption, as there are other means to state something like this (e.g. two elements can be explicitly defined as being different).

As explained in the section 2.2.3, OWL 2 is based on DLs. Since this is a fragment of FOL, it inherits many of the properties of that logic. In particular, it inherits the OWA and the non UNA. This has led to the definition of the interesting sublanguage of OWL 2 DL.

2.2.4 OWL 2 and rules

Importance of rule-based systems, together with the will to extend the OWL 2 expressive power has led to the combination of OWL 2 ontologies with rule-based knowledge representation and reasoning. Main obstacle that had to be overcome was the fact that OWL 2 adheres to the OWA, in contrast to rules, that they generally follow the CWA. SWRL is a rule extension that follows the OWA and thus is entirely in the spirit of the OWL 2. SWRL adds to the expressive power of OWL 2 DL by allowing the modelling of certain axioms which lie outside the capability of OWL 2 DL (Hitzler and Parsia 2009).

SWRL contains OWL 2 DL as a proper part; that is, all OWL 2 DL axioms are SWRL axioms. A SWRL knowledge base may contain a set of rules, which consist of a body, and a head which themselves are sets of SWRL atoms.

A SWRL atom may be of the following forms:

- Unary atoms: C(arg1), where C is an OWL 2 class expression
- Binary atoms: P(arg1, arg2), where P is an OWL 2 object property

Arguments are considered to be OWL 2 individuals, and a SWRL rule is a sequence of atoms.

Incapability of OWL 2 to support cyclic definitions is an example of how rules can enrich its expressive power. SWRL can extend OWL 2 expressive power by allowing cyclic definitions, i.e. cyclic dependencies between the defined names in the set of concept definitions (Baader, Horrocks and Sattler, Description Logics 2009). Major role to this direction play the "built-ins" axioms. "Built-ins" are atoms with a fixed and predefined interpretation. Cyclic dependencies could be expressed with the use of the built-in axiom *sameAs*. The SWRL submission includes built-in axioms for value comparison, mathematics, and string manipulation among others.

2.2.5 Protocol and RDF Query Language (SPARQL)

SPARQL (W3C 2008) query language was first standardised in 2008 by the W3C for querying semantic web data. However, only the simple semantics of RDF are supported by SPARQL 1.0, which does not allow any reasoning (Kollia, Glimm und Horrocks 2011).

SPARQL in its currently recommended version it is considered to be an appropriate language for specifying queries that return only explicitly defined knowledge within an ontology. Thus, SPARQL cannot compute knowledge and deduct to implicit knowledge so as to define queries upon it.

Although there is not yet a standardised query language for OWL knowledge bases, the majority of currently used semantic reasoners do support a query language. Pellet reasoner supports SPARQL – DL, which is a subset of SPARQL. On the other hand, TrOWL reasoner supports SPARQL but is restricted only to ABox queries.

SPARQL 1.1 outstands from the previous version because it includes entailment regimes. This allows for using SPARQL also as a query language over OWL ontologies with query answers also including solutions that are not explicitly stated, but implicit consequences of the queried ontology.

2.2.6 Semantic reasoners

A semantic reasoner serves as an inference engine as it allows inferring/deriving implicit knowledge from the explicitly stated one. Reasoning could be split into TBox and ABox reasoning. TBox reasoning is about classifying the classes among the class hierarchy. On the other hand, ABox reasoning refers to the assignment of individuals to certain classes among the class hierarchy and checking about inconsistent assignments. ABox reasoning is usually activated after TBox reasoning. Reasoning is important both to ensure the quality of an ontology, and in order to exploit the rich structure of ontologies and ontology based information.

Currently, there is a number of reasoners that can be used as an effective inference engine for inferring implicit knowledge. Within this underlying work the following three are going to be used for experimental reasons:

Pellet is a DL OWL reasoner written in java, and it provides a wide variety of reasoning services for OWL ontologies. Pellet includes support for OWL 2 profiles like DL and EL. The expressiveness DLs that are supported by Pellet is $\mathcal{SROIQ}^{(\mathcal{D})}$ and is released under a dual license model (Sirin, et al. 2007).

Hermit is a DL OWL reasoner developed by Oxford University Computing Laboratory. It is released under LGPL (GNU Lesser General Public License) and is considered to be fully compatible with OWL 2 (Shearer, Motik and Horrocks 2008).

TrOWL reasoner was developed to provide reasoning support for large ontology-based knowledge bases and was used to support the work on the MOST project[1]. TrOWL reasoner offers support for all the expressive power of OWL 2 DL (Thomas, Pan and Ren 2010).

2.2.6.1 Semantic Reasoners services

In order for a reasoner to infer implicit knowledge from the knowledge explicitly stated within an ontology, or to ensure its high quality, the following reasoning services are used (Aßmann, et al. 2013):

- *Consistency checking* validates ABox against TBox and returns *true* in case the former is consistent in regard to the latter.
- *Classification* classifies ABox to TBox through the inference process.
- *Satisfiability checking* finds all the empty classes of an ontology (classes that cannot contain any individual) and returns *false* in case that one class expression is not satisfiable.
- *Subsumption checking* arranges ABox into classes or categories and returns *true* if an individual is an instance of a class expression.
- *Find unsatisfiability* searches for all unsatisfiable concepts and returns a set of them.
- *Axiom explanation* returns a set of axioms.
- *Inconsistency explanation* returns a set of axioms for each inconsistency.
- *Query answering* returns an answer set for a given query.

[1] http://www.most-project.eu, last accessed 30.05.2012

2.3 Related Work

It has been shown that defining a metamodel is one of the most crucial tasks for defining a correct and complete modelling language and that it outstands during this process. There has been an extended research regarding related work concerning metamodel definition, and definition of its static semantics.

In the following section, related approaches are grouped into two main categories: approaches that try to formalize static semantics related to specific meta-metamodels, and approaches concerning constraint languages used within metamodels that could potentially be used for standardizing and formalizing definition of meta-metamodel static semantics.

2.3.1 Meta-metamodel - oriented approaches

MOF is considered to be one of the most popular meta-metamodels. Technologies standardized by OMG like UML, MOF, CWM, SPEM, XMI use MOF for their definition. As it is already mentioned, OCL is mainly used for defining static semantics within metamodels (e.g. UML). However, there have been a number of attempts to use OCL with both metamodels (e.g. UML) and MOF. In (Boronat and Meseguer 2007) there has been an attempt to formalize semantics definition of MOF, despite its genericity, since many different metamodels rely on the correctness of the MOF.

Another meta-metamodel suggested by EMF (Eclipse Modelling Framework) is ECore (Steinberg, et al. 2009). (Petrascu and Chiorean, Proposal of a Set of OCL WFRs for the ECore 2009) and (M. Garcia 2007) are proposals for a set of OCL well-formedness rules for the ECore meta-metamodel.

The same authors, in (Petrascu and Chiorean, Towards Improving the Static Semantics of XCore 2010) try to analyse the state of facts concerning the static semantics of the XCore meta-metamodel and they propose improvements in formalizing it.

Both approaches come to the conclusion that formalizing static semantics definition of meta-metamodel is a necessity and their goal is to come up with an identification of a “core” set of constraints used by all meta-metamodels.

2.3.2 Constraint languages - oriented approaches

In (Garcia, et al. 2006), the study of the validations needed to carry out such a semantic analysis, and the development of a semantic validation tool is presented, which can be used for implementation of PMIF (Performance Model Interchange Format) import/export mechanisms. For this purpose the OCL language is used.

There have been a number of studies concerning OCL used as an expression language to describe a set of well-formedness rules and semantic constraints for considered models and

implementation of tools for validating these constraints. Focus of these studies lies on formalizing static semantics definition and their validation within metamodel. (Loecher and Ocke 2004) and (Cadavid, Baudry and Combemale 2011) take the previous approaches one step further by exploring the conjunct use of MOF metamodel and OCL.

The use of Alloy constraint language for defining formal semantics of a modelling language has been proposed in (Kelsen and Ma 2008). This study once again emphasizes the importance of formal static semantics for reasoning about the modelling language and for providing tool support, and the advantages of using Alloy constraint language due to its low notation complexity and automatic analysability.

A recent study (Zedlitz, Jörke und Luttenberger 2012) has proposed the transformation of UML class diagrams into OWL 2 ontologies with the use of QVT transformation language, both metamodels of which (UML and OWL respectively) are based on MOF meta-metamodel. (Walter, Parreiras and Staab 2009) reports on a novel approach that allows the use of ontologies to describe Domain Specific Languages (DSLs). According to this approach the formal semantics of OWL together with reasoning services allows constraint definition, progressive evaluation, suggestions, and debugging. Additionally, this approach integrates existing metamodels, concrete syntaxes and a query language.

Our approach is a proposal for formalizing the definition of semantic constraints of ADOxx meta-metamodel, which currently are defined with the use of C++ programming language. Main requirements for standardising the definition of static semantics are to specify the language used for formally defining the semantic constraints, and the engine that will execute the constraints so as to identify constraint violations. Within our approach, we will use the expressive OWL 2 ontology language as a means to formalise ADOxx semantic constraints within meta-metamodel. The use of a semantic reasoner (e.g. Pellet, HermiT, TrOWL etc.) will be used as the engine to execute the constraints and visualise their violations.

3 Building ADOxx Meta-metamodel Ontology

Within this chapter, we are going to describe extensively the process of mapping ADOxx meta-metamodel to ontology main components with the use of OWL 2 ontology language. The ADOxx meta-metamodel is going to be decomposed into its main core concepts. Main objective is to define a complete and systematic mapping approach.

The first subsection of this chapter describes the mapping approach regarding ADOxx meta-metamodel core elements, i.e. a library, a model type, an attribute etc. The second subsection refers to the relationships that appear within the meta-metamodel and establish the relations among ADOxx core elements, and how they can be defined with the use of OWL 2. The third subsection focuses on the mapping approach that was followed for defining properties of ADOxx core elements that can be assigned to them during the customization of a metamodel. The fourth subsection refers to the ADOxx meta-metamodel system elements, how they were integrated within the ontology, and to the issues stemming from OWL 2 and its basic concepts. The last section gathers all conclusions and lessons learned during this process.

3.1 Main Mapping Approach

After a thorough study of the main components that an ontology consists of and the purpose that they serve, we tried to formulate a main mapping approach, according to which the ADOxx meta-metamodel would be converted into an ontology.

As it is already mentioned in section 2.2.1, TBox represents general concepts and the relations between these concepts. Thus, it could be considered as a static ontology part, which does not change rapidly or too often. On the contrary, ABox can continuously or rapidly be changing, by adding or removing instances of the concepts, or by changing the interrelations between the instances of the concepts. Correspondingly, ADOxx meta-metamodel, is considered to be a static schema, which was not built to change often over time, in contrast to ADOxx metamodels, are based on the ADOxx meta-metamodel and are more frequently edited or customized.

The Figure 3-1 represents the main mapping approach that is going to be followed in order to convert ADOxx meta-metamodel to an ontology:

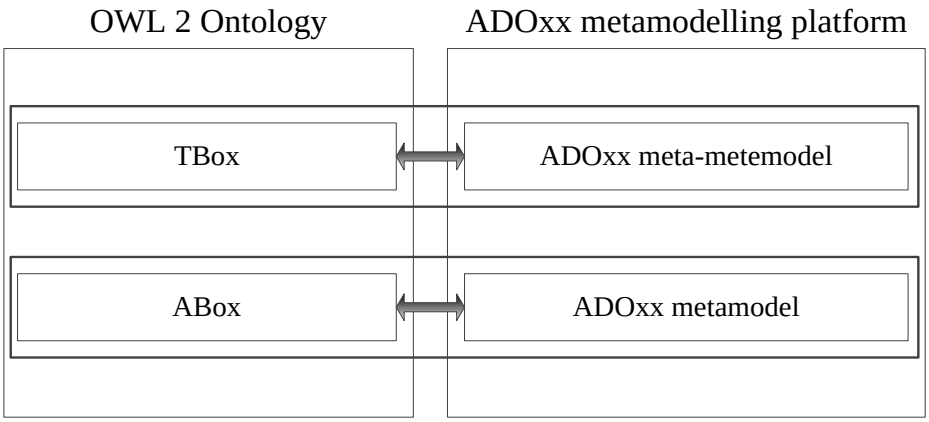


Figure 3-1: Converting ADOxx meta-metamodel into an ontology - main mapping approach (conceptualization)

The mapping approach that was defined and followed for the conversion of ADOxx meta-metamodel into an OWL 2 ontology is summarized as followed:

- An ADOxx core element is always represented as an OWL 2 class
- An ADOxx core relation which expresses:
  - *Aggregation* is represented as an OWL 2 object property
  - *Association* is represented as an OWL 2 object property
  - *Generalization* is expressed within OWL 2 class taxonomy
- An ADOxx core element property is represented as an OWL 2 class
- An ADOxx system core element, which can be a specialization of the ADOxx core elements “Attribute”, “Modelling class” or “Relation class” is always represented as an OWL 2 class

Main prerequisite for the definition of the main mapping approach was the will to stay as close as possible to the philosophy of ADOxx metamodelling platform, as far as method engineer's freedom to build new and customize already existent metamodels are concerned.

3.1.1 Mapping ADOxx core elements

In ADOxx meta-metamodel as depicted in section 2.1.3, the ADOxx elementary constructs can be easily identified. According to the mapping approach defined, every ADOxx core element is mapped to an OWL 2 class within ontology TBox.

An advisable visualization principle is to organize an ontology TBox by its class hierarchy. Class hierarchy is a directed acyclic graph (DAG). A hierarchical class structure is a semantic organization of an ontology, so as to gain a holistic sense of an ontology when looking at the top levels of the hierarchy (Tu, et al. 2005). A hierarchical class structure provides a better understanding of the ontology and the concepts represented within its TBox. Moreover, it provides the ability to exploit the advantages that come with its use, regarding inheritance of properties from a super class to its subclasses.

Based on the conceptual view of the ADOxx meta-metamodel as depicted in Figure 2-3, information provided regarding class hierarchy is not considered to be adequate. On the contrary, Figure 3-2 depicts the design view of ADOxx meta-metamodel and clarifies the hierarchy among ADOxx core elements, making it easy to build ontology TBox, in such a way, that it allows a good understanding and an easy maintenance.

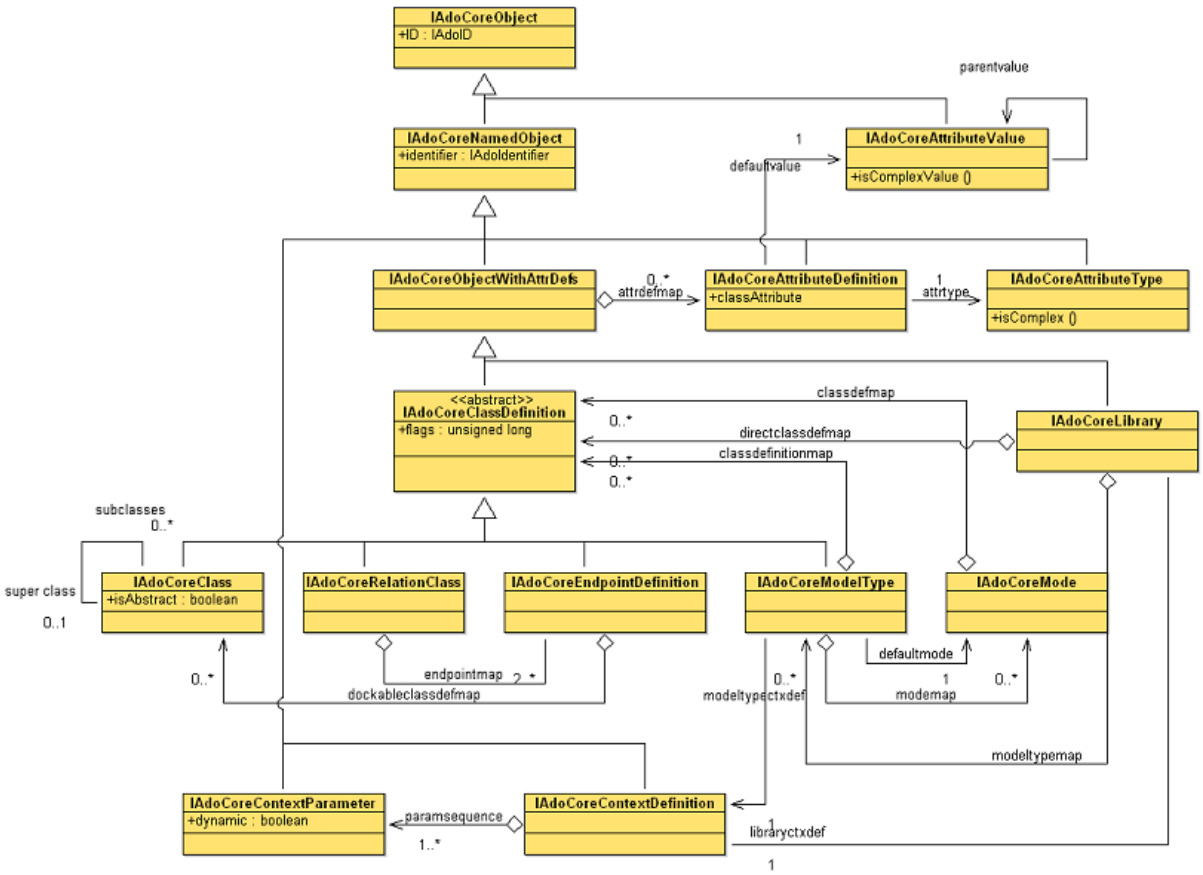


Figure 3-2: ADOxx meta-metamodel - design view (BOC IS GmbH n.d.)

Taking into consideration the importance of the class hierarchy, we decided to make use of two classes, which do not represent concrete ADOxx core elements, and thus are considered to be abstract classes: “NamedElement” and “ElementWithAttributes”.

Every ADOxx core element that contains the attribute “Name” and can have a language specific name is represented as a subclass of the abstract class “NamedElement”. Furthermore, every ADOxx core element to which at least one attribute can be assigned is represented as a subclass of the abstract class “ElementWithAttributes”. Any other ADOxx core element like for instance “Attribute value”, or “Language” lies at the same hierarchy level of that of the two mentioned abstract classes.

Apart from the class hierarchy, *DisjointClasses* class axiom within OWL 2 class definition is used to define explicitly which instances cannot be members of two or more classes simultaneously. An ADOxx specific example would be that a relation class cannot be a library, a model type or an attribute at the same time (given that every element has a unique language independent identifier).

The Figure 3-3 represents TBox class hierarchy of ADOxx core elements defined as OWL 2 classes.

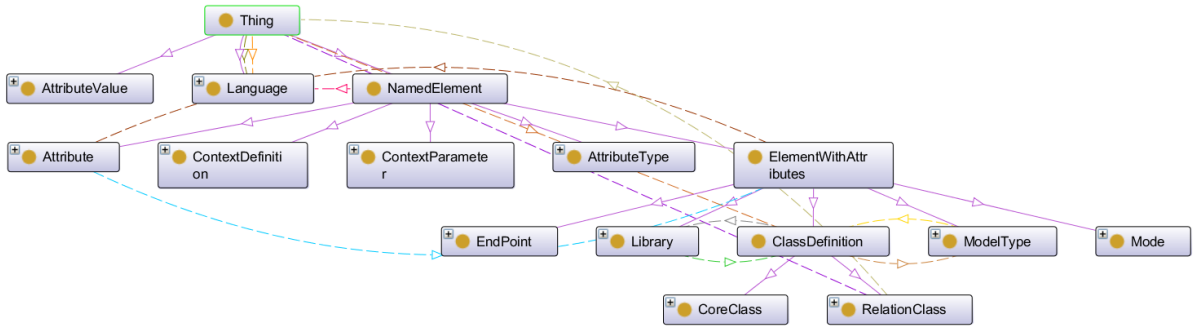


Figure 3-3: Ontology class hierarchy - ADOxx meta-metamodel core elements

3.1.2 Mapping ADOxx core relations

In ADOxx meta-metamodel, three types of relationships can be found: aggregation, association and generalization (Figure 3-4).

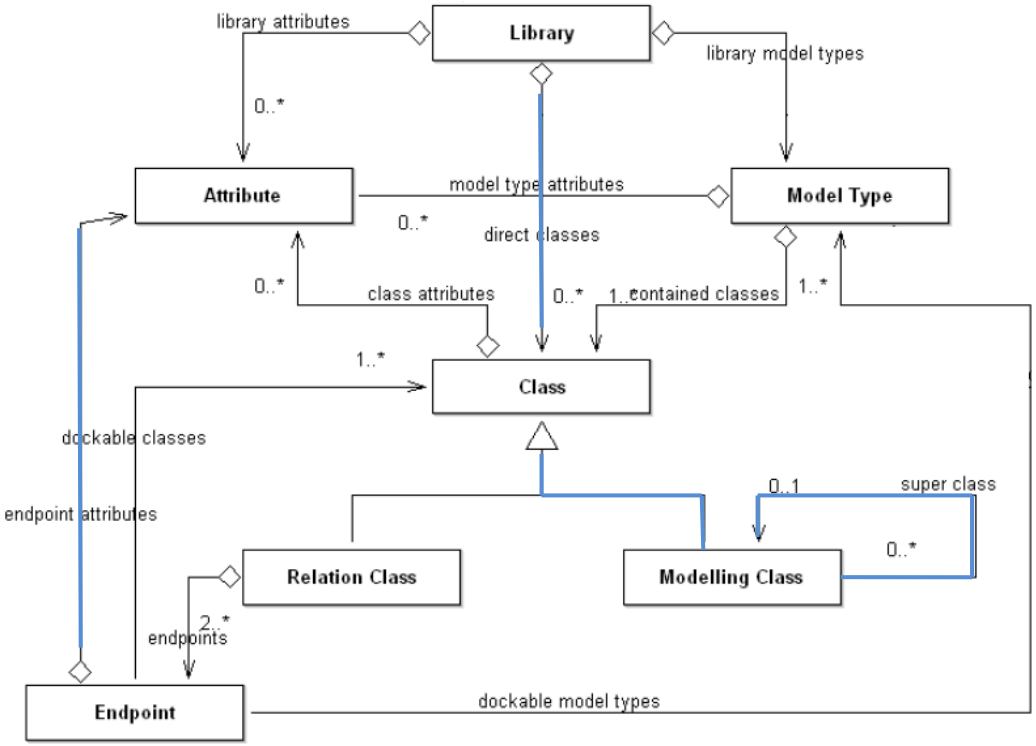


Figure 3-4: Example of ADOxx meta-metamodel core relations

According to the defined mapping approach, two main guidelines apply to the mapping of ADOxx core relations. Every ADOxx core relation that expresses the notion of “aggregation” or of “association” is mapped to an OWL 2 object property. The second guideline implies that every ADOxx core relation that expresses the notion of “generalization” is expressed through the TBox class hierarchy.

Within the next three subsections, every type of relationship is going to be extensively described.

3.1.2.1 Aggregation

The concept of aggregation indicates that an instance of a class contains a set of instances of another class. It is a typical whole/part relationship. Aggregation defines an asymmetric (Booch, Jacobson and Rumbaugh 1997), and an irreflexive (An Oracle White Paper, Getting Started With UML Class Modeling 2007) relationship. In case that an aggregation is found as a reflexive relationship, this indicates that an instance of a class can be part of *another instance* of the *same class*, but not of itself.

To every ADOxx core relation that expresses the notion of "aggregation", the object property characteristics *AsymmetricObjectProperty* and *IrreflexiveObjectProperty* are assigned, as an aggregation is by default asymmetric (Gogolla and Richters 1998) and irreflexive.

An ADOxx specific example of a relation that corresponds to an aggregation would be that a class can have one or more attributes (Code snippet 3-1).

```
ObjectProperty: <#hasAttribute>

    Characteristics:
        Asymmetric,
        Irreflexive

    Domain:
        <#ElementWithAttributes>

    Range:
        <#Attribute>

    InverseOf:
         <#attrBelongsTo>
```

Code snippet 3-1: ADOxx "Class attributes" relation as OWL 2 object property

The "Class attributes" relation is defined as *asymmetric* and *irreflexive*. The two examples below (Figure 3-5, Figure 3-6) explain the use of these characteristics.

Example 3-1: "Class attributes" relation - *asymmetric*

```
In case that the core class "A" has the attribute "a", it would be wrong to be assumed that
the attribute "a" has the attribute "A", since "A" is not an attribute, but a core class
```

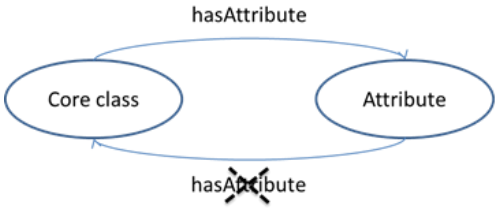


Figure 3-5: Example of an *asymmetric* ADOxx core relation

Example 3-2: “Class attributes” relation - *irreflexive*

```
The core class “A” can have the attribute “a”, but attribute “a” cannot have another
attribute.
```

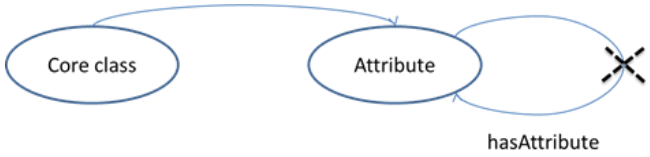


Figure 3-6: Example of an *irreflexive* ADOxx core relation

3.1.2.2 Association

Association indicates that there is some relationship between instances of the classes, without any hint of whole/part relationship. In addition, instances can have cyclic relationships, something that does not happen with aggregation.

An example of an ADOxx core relation that corresponds to an association is the “Super class” relation, which defines that a class can have none, or exactly one direct super class. In this case, the relationship is considered to be a cyclic association. This means that an instance of this class can be related to other instances of the same class, but not to itself. This is the reason why the corresponding object property is not defined as reflexive.

```
ObjectProperty: <#isSuperClassOf>

    Characteristics:
        Transitive

    Domain:
        <#CoreClass>

    Range:
        <#CoreClass>

    InverseOf:
        <#isSubClassOf>
```

Code snippet 3-2: ADOxx “Super class” relation as OWL 2 object property

Moreover, the characteristic *TransitiveObjectProperty* is assigned to the object property “Super class”. The following example clarifies the use of this characteristic.

Example 3-3: ADOxx “Super class” relation - *transitive*

```
In case that the core class ”A” is super class of the core class “B”, and the core class “B”
super class of the core class “C”, then it is assumed that core class “A” is also super class
of the class “C”.
```

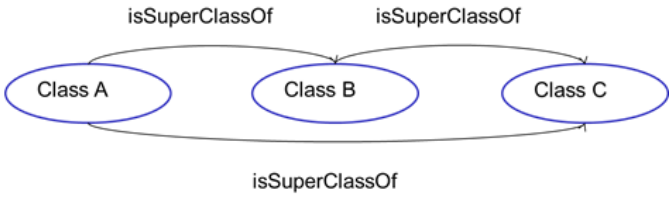


Figure 3-7 Example of a *transitive* ADOxx core relation

3.1.2.3 Generalization

Generalization is a relationship between two classes that indicates that one of them has an "is a kind of" semantics from one class to another. It also indicates property inheritance. An ADOxx specific example would be that a class can either be a relation class, or a modelling class, and both of them can have class attributes, as both are classes. This relationship, in contrast to the others, is not represented as an OWL 2 object property, but is expressed within the class hierarchy, defining for instance, that classes “Modelling class” and “Relation class” are subclasses of the super class “Class Definition”.

```
Class: <#RelationClass>

    SubClassOf:
        <#ClassDefinition>

    DisjointWith:
        <#CoreClass>
```

Code snippet 3-3: ADOxx generalization relationship

3.1.2.4 Additional remarks

After an extensive description of the three types of ADOxx core relations and the followed mapping approach, it would be thoughtful that we mention some additional general remarks.

Object properties defined always contain the following two axioms: the *ObjectPropertyDomain* and the *ObjectPropertyRange* axiom. These two axioms are used to restrict the source and the target class respectively that the individuals connected through this object property are members of.

Furthermore, the *InverseObjectProperty* and *ObjectPropertyChain* axioms are used selectively for simplifying the construction of *EquivalentClass* axioms for reasoning purposes.

An ADOxx specific example of a core relation mapped to an inverse object property would be the object property “attrBelongsTo”, which is defined as the inverse object property of the ADOxx relation “Class attributes”.

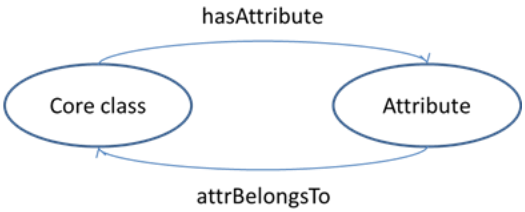


Figure 3-8: Example of an *inverse* OWL 2 object property - ADOxx core relation "Class attributes"

Regarding the *ObjectPropertyChain* axiom, an ADOxx specific example of a core relation expressed as an object property chain would be the case represented in Figure 3-9.

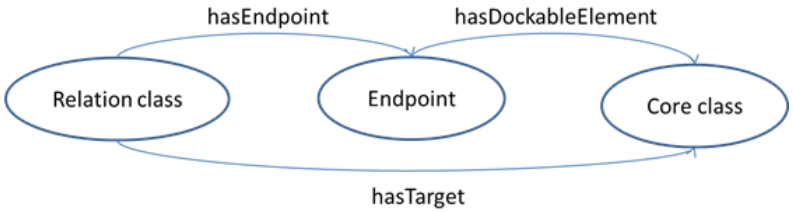


Figure 3-9: Example of an *object property chain* of ADOxx core relations

The Figure 3-9 describes that in case that a relation has an endpoint, which has a core class as dockable element, consequently this core class is considered to be one of the two targets of this relation.

3.1.3 Mapping ADOxx core element properties

The definition of every ADOxx meta-metamodel core element is accompanied by a set of properties that can be assigned to it. Within the following sections, we are going to introduce the mapping approaches that we defined while trying to integrate these core element properties into the ontology.

Studying the ADOxx metamodelling platform showed that the majority of property combinations are enforced by the metamodelling platform. Taking this into consideration, we came up with a number of possible mapping approaches, before defining the final one.

At this point should be mentioned that, for brevity and consistency, the following sections focus only on the properties regarding the ADOxx core class element.

3.1.3.1 Core element property combinations as OWL 2 classes

The first mapping approach was based on the concept that every allowed by the metamodel editor combination of core element properties is mapped to an OWL 2 class. The Figure 3-10 represents an excerpt of the ontology TBox, regarding the representation of the ADOxx core class property combinations as OWL 2 classes.

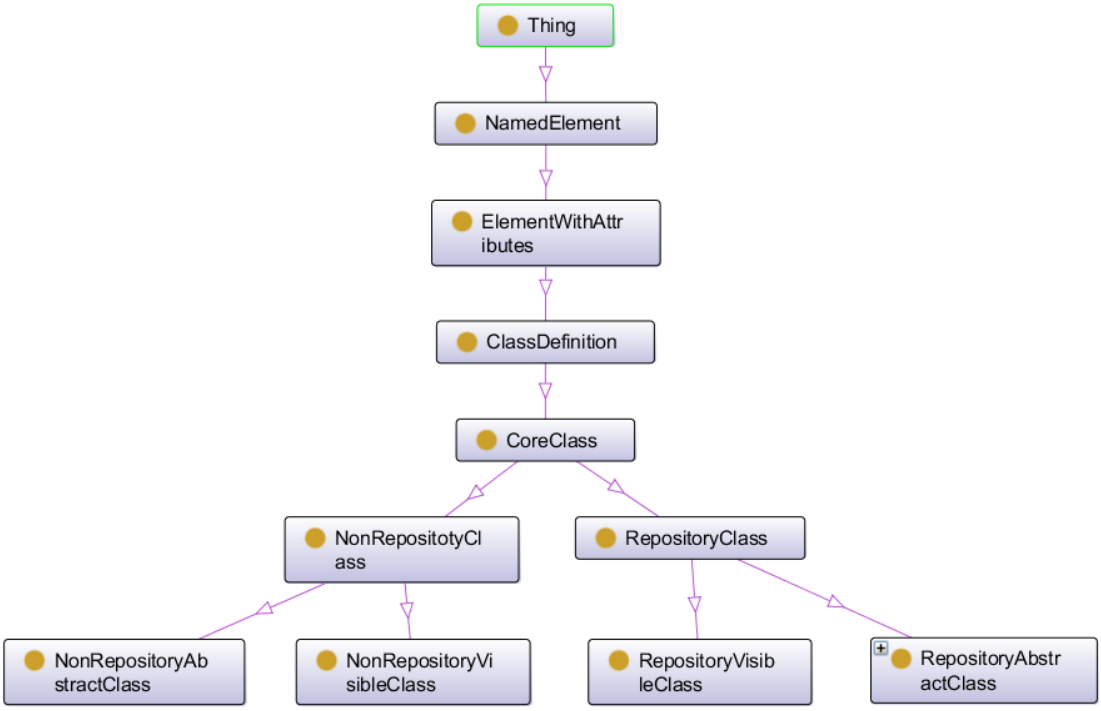


Figure 3-10: ADOxx core class property combinations as OWL 2 classes

According to this mapping approach, an instance of an ADOxx core class that is defined as *repository* and *abstract* is member of the OWL 2 class "RepositoryAbstractClass" (Code snippet 3-4).

```
Individual: <#class_a>

    Types:
        <#CoreClass>,
        <#RepositoryAbstractClass>
```

Code snippet 3-4: ADOxx core class individual definition – *repository* and *abstract*

Main advantage of this approach is the fact that minimizes the risk of defining instances of ADOxx core elements with the wrong property combination.

However, the big amount of the allowed ADOxx core element property combinations would lead to a bloated ontology TBox. This would lead to a hard to maintain and update TBox. More specifically, the majority of ADOxx core element properties can take two values; either true (assigned), or false (not assigned). As it was mentioned in the section 2.2.3.2, OWL 2 is based on the OWA, which states that *everything is true, unless it is explicitly stated that something is false.* This means that under no circumstances can be assumed that an individual that corresponds to the core class, and which is not a member of the class "RepositoryClass" is not a repository core class instance. Thus, for every core element property two classes have to be defined – the property itself, and the negation of the property.

3.1.3.2 Core element properties as OWL 2 data properties

The drawbacks of the first mapping approach regarding the ADOxx core element properties led us to search for an alternative one. The second mapping approach makes use of the OWL 2 data properties. For every ADOxx core element property, an OWL 2 data property is defined. The Figure 3-11 shows the definition of core class properties as OWL 2 data properties.

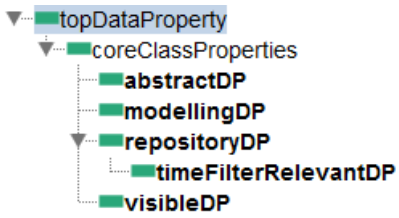


Figure 3-11: ADOxx core class property definition as OWL 2 data properties

According to this approach, every ADOxx core element property is mapped to an OWL 2 data property. *DataPropertyRange* axiom is used to restrict the literals pointed to by the property to be in the specified unary data range, which in this case is "boolean". Consequently, two values can be assigned - true or false. Additionally, every data property is defined as *functional*, preventing the case of a data property having the values "true" and "false" simultaneously.

The Code snippet 3-5 shows the definition of the ADOxx core class property *abstract* as an OWL 2 data property.

```
DataProperty: <#abstractDP>

    Characteristics:
        Functional

    Domain:
        <#CoreClass>

    Range:
        xsd:boolean

    SubPropertyOf:
        <#coreClassProperties>

    DisjointWith:
        <#visibleDP>,
        <#timeFilterRelevantDP>
```

Code snippet 3-5: ADOxx core class property - *abstract* as data property

In the case of an ADOxx core class instance, that is defined as *repository and not abstract*, the only action that has to be done is to assign these data properties to the instance of the OWL 2 class „CoreClass“ and assign to it the correct boolean value (true/ false) (Code snippet 3-6).

```
Individual: <#class_a>

    Types:
        <#CoreClass>

    Facts:
     <#visibleDP>  false,
     <#abstractDP>  true,
     <#repositoryDP>  true,
     <#timeFilterRelevantDP>  false
```

Code snippet 3-6: ADOxx core class individual definition – *repository* and *abstract*

Main advantage of this approach is the fact that simulates pretty accurately the way that properties are assigned to ADOxx core elements within ADOxx metamodelling platform. However, for reasons that are going to be mentioned, but not extensively explained at this point, we decided to explore another alternative mapping approach for ADOxx core element properties. The first reason was reasoning performance and efficiency, which was seriously affected by the extended use of data properties. Moreover, poor expressive power of inconsistency explanation in case of a wrong combination of data properties assigned to an individual that represents an ADOxx core element was the second reason that led us to this decision.

3.1.3.3 Core element properties as OWL 2 classes

The drawbacks of the two previous mapping approaches led us to the third and final one regarding ADOxx core element properties. This approach could be characterized as a hybrid of the two previous ones, as it combines main features of both of them. According to this approach, every ADOxx core element property is mapped to an OWL 2 class. Mutually exclusive properties are implemented with the use of *disjointProperty* axiom. In addition, sub-properties are implemented within the class hierarchy. The figure below (Figure 3-12) represents the definition of ADOxx core class properties as OWL 2 classes.

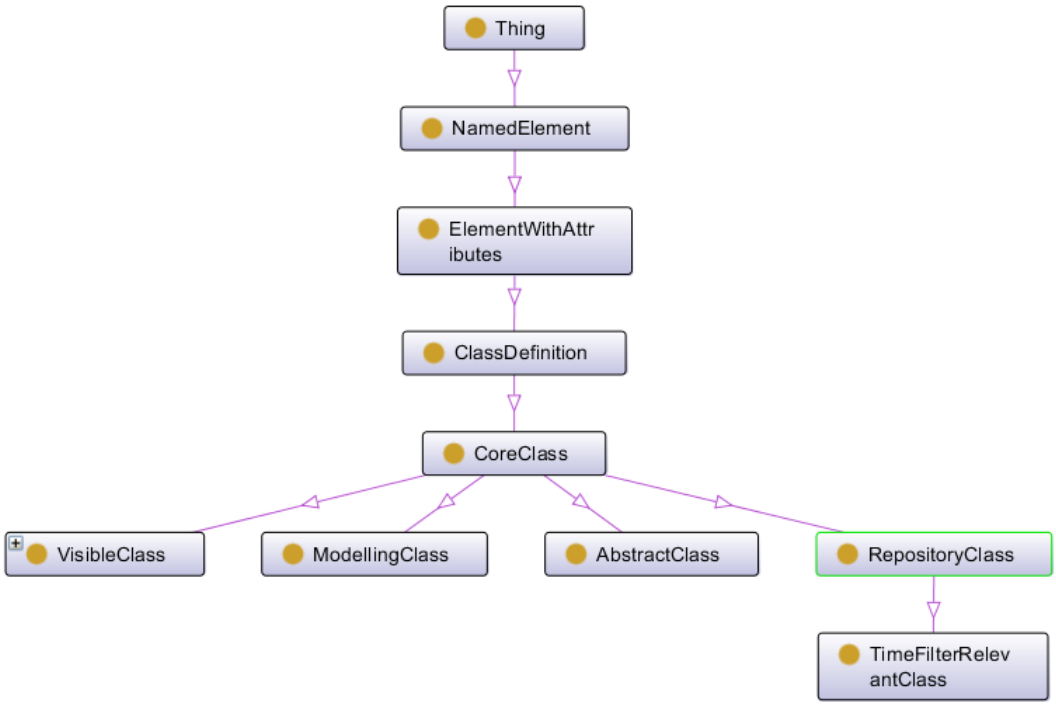


Figure 3-12: ADOxx core class property definition as OWL 2 classes

Having mapped the ADOxx core element properties, in case of an OWL 2 individual of the class "CoreClass", which represents an ADOxx core class instance, which is, for instance, *repository and not visible*, the only action that has to be done is to make this individual member of the OWL 2 classes that represent the desired ADOxx core element properties (Code snippet 3-7).

```
Individual: <#class_a>

    Types:
        not (<#VisibleClass>),
        <#CoreClass>,
        <#AbstractClass>,
        not (<#TimeFilterRelevantClass>),
        <#RepositoryClass>
```

Code snippet 3-7: ADOxx core class individual definition – *repository, abstract and not visible*

Keeping in mind that OWL 2 sticks to the OWA, the previous example shows a pretty usual case that should be taken into account. In the case of an ADOxx core class instance that is *not visible,* it has to be explicitly stated that the individual of the OWL 2 class "CoreClass" is *NOT* member of the OWL 2 class "VisibleClass".

Main advantage of this approach is that problems that appeared before, like a not effective reasoning performance and a poor expressive power of inconsistency explanations were overcome. Additionally, this approach remains pretty close to the way that properties are assigned to a core element within ADOxx metamodelling platform, without constraining the method engineer, but without making it easier for him to assign a wrong combination of properties to a core element.

3.1.4 Mapping ADOxx system elements

ADOxx system elements are predefined ADOxx metamodel elements. ADOxx system elements are metamodel independent; they can be reused in any metamodel and are needed by platform components such as the model editor, the repository etc.

An ADOxx specific example could be the following one:

The ADOxx system attributes POSX and POSY must be assigned to an ADOxx core class definition for being able to place an instance of this core class among the model editor.

Within ADOxx metamodelling platform, the following pattern is followed concerning the definition of ADOxx core elements. A core element, like for instance, an attribute, is once defined in the global container and used multiple times by other ADOxx core elements (Figure 3-13).

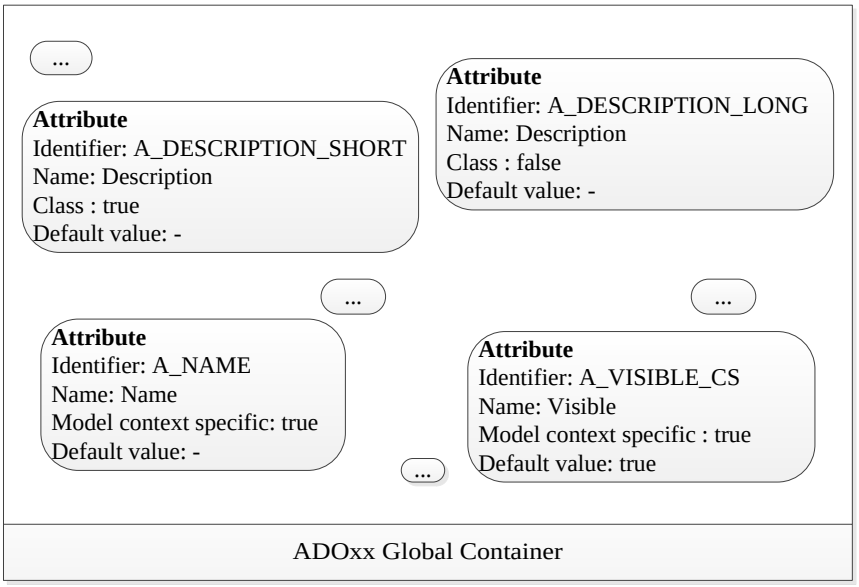


Figure 3-13: ADOxx global container regarding attribute definitions

In the case of an attribute definition within the ADOxx global container, this attribute can be assigned to multiple metamodel class instances. However, its default value may vary depending on the owner class. The Figure 3-14 represents the assignment of the attribute "Visible" to two different classes.

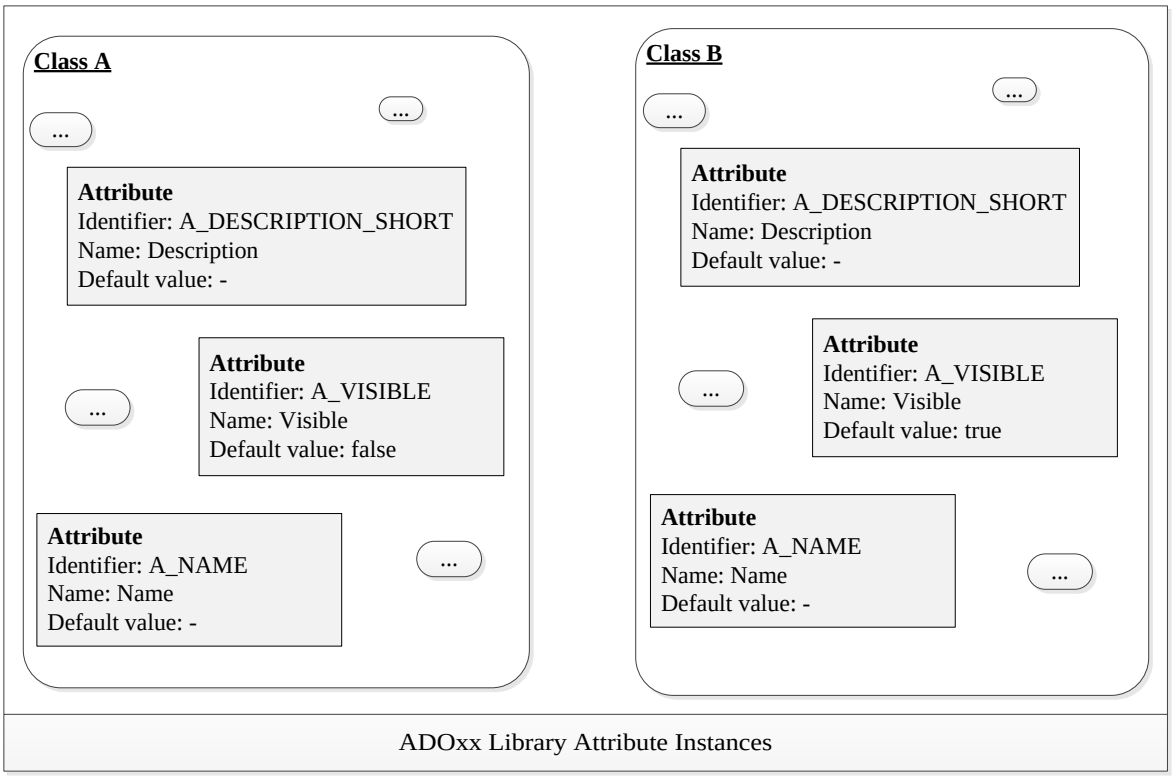


Figure 3-14: ADOxx library attribute instances

As the figure above reveals, both classes contain the same attribute. However, default value of the attribute assigned to the first class is “true”, in contrast to the second class, in which the attribute default value is “false”.

The example above clarifies the use and importance of ADOxx system elements. In addition, the fact that semantic constraints upon the metamodel are tightly coupled with the use of these system elements, led us to the decision to include them in the definition of the mapping approach.

Within the following subsections, two mapping approaches regarding the ADOxx system elements are going to be extensively described.

3.1.4.1 ADOxx system elements as OWL 2 individuals

The first mapping approach can be summarized as follows: For every ADOxx system element defined within the ADOxx global container, an ABox individual is defined. For every instance of this ADOxx system element owned by an ADOxx core element, an additional individual is created, which is defined as same with the one that represents the system element in the ADOxx global container. An ADOxx system element can be a core class, a relation class or an attribute.

The Figure 3-15 represents diagrammatically the first mapping approach, regarding system attributes. The same approach was applied for system core classes and relations; thus everything described within this section applies for all types of ADOxx system elements.

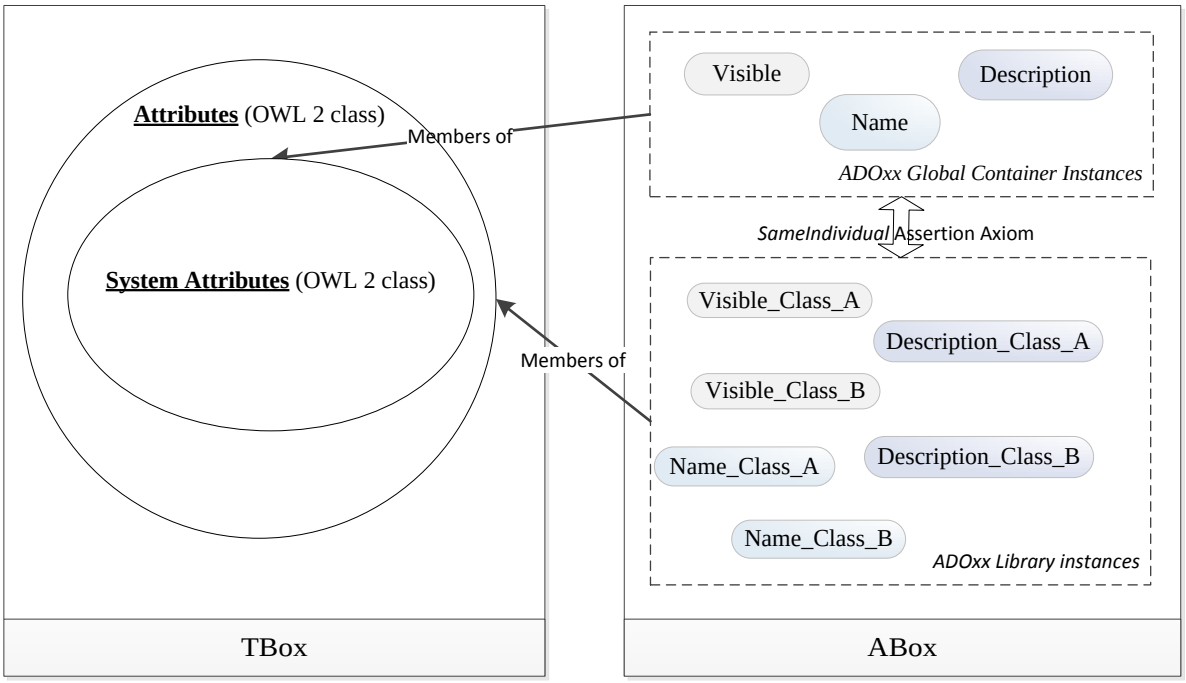


Figure 3-15: ADOxx system attributes - global container and library instances, represented as OWL 2 individuals

As we can see in the diagram above, ADOxx system attributes are considered to be individuals of the OWL 2 class "SystemAttributes" within ontology TBox. This OWL 2 class is defined to be a subclass of the "Attributes" OWL 2 class, because system attributes are a special type of attributes. Every system attribute defined within ADOxx global container is mapped to an ABox individual, member of the former OWL 2 class. Correspondingly, every instance of the system attributes assigned to an ADOxx core element is mapped to an individual, member of the latter OWL 2 class. The semantic equivalence between an ADOxx global container system attribute and its assigned to an ADOxx core element instances is expressed with the use of *SameInidividuals* axiom. With this axiom, it is explicitly stated that every owned instance of this system attribute is semantically exactly the same with this system attribute.

The *SameIndividual* axiom states that two or more individuals are considered to be equivalent and identical. According to (Halpin, et al. 2010), there is a number of varieties of identity and similarity. The uses of the axiom that can be found to be inconsistent with its strict logical definition can be summarized as follows:

- *Identical but referentially opaque*: when two individuals do identify to the same thing, but all the properties ascribed to one individual are not necessarily the same or appropriate for the other.
- *Similar*: when two different individuals share some but not all properties in their given incomplete description.
- *Related*: when two individuals share no properties in common in a given description, but are nonetheless closely aligned in some fashion.

Having described the uses of the axiom that violate its strict definition, it would be wise to infer that our case fits the first use. Although ADOxx system attributes and their assigned instances denote the same individual, they are considered to be referentially opaque.

The Figure 3-16 represents diagrammatically use of the axiom and how the OWL 2 individuals are considered to be referentially opaque.

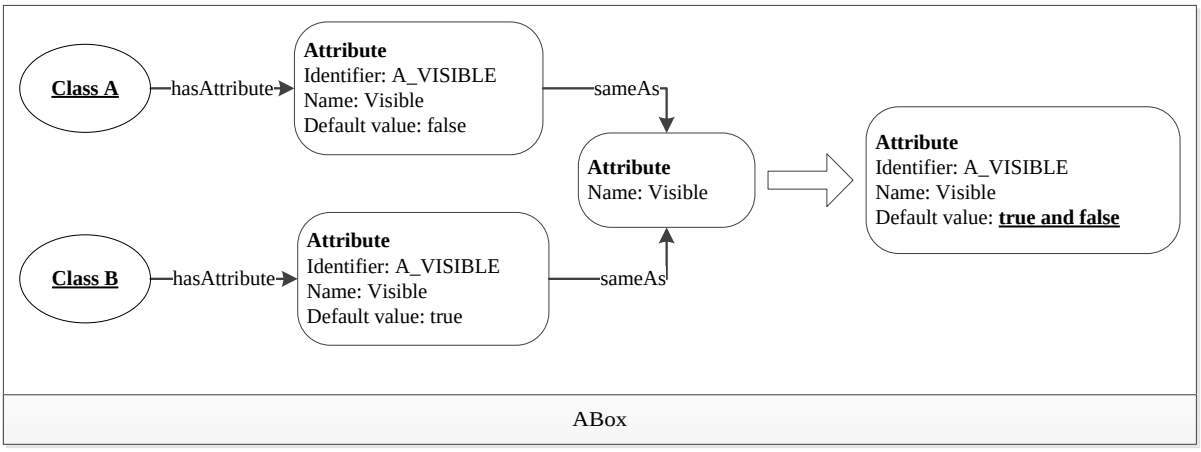


Figure 3-16: Side effects of the use of the *SameIndividual* axiom

These side effects cause inferential inconsistency by leading to wrongly inferred class memberships.

The code snippets below represent the definition of this scenario in OWL 2.

```
Individual: <#visible>

    Types:
        <#SystemAttribute>

    SameAs:
        <#visible_class_A>,
        <#visible_class_B>
```

Code snippet 3-8: ADOxx global container instance of attribute "Visible"

```
Individual: <#visible_class_A>

    Types:
        <#Attribute>,
        <#ModelContextSpecificAttribute>

    SameAs:
        <#visible>

Individual: <#visible_class_B>

    Types:
        <#ModelContextSpecificAttribute>,
        <#Attribute>

    SameAs:
        <#visible>
```

Code snippet 3-9: ADOxx library instance of attribute “Visible”

```
Individual: <#class_A>

    Types:
        <#CoreClass>

    Facts:
     <#hasAttribute>  <#visible_class_A>

Individual: <#class_B>

    Types:
        <#CoreClass>

    Facts:
     <#hasAttribute>  <#visible_class_B>
```

Code snippet 3-10: Assignment of attribute “Visible” to two different classes

After the reasoning process the following wrong class memberships are inferred (Code snippet 3-11)

```
Individual: < #class_A>

    Types:
        <#CoreClass>

    Facts:
     <#hasAttribute>  <# visible_class_A >,
     <#hasAttribute>  <# visible >,
     <#hasAttribute>  <# visible_class_B >

Individual: < #class_B>

    Types:
        <#CoreClass>

    Facts:
     <#hasAttribute>  <# visible_class_A >,
     <#hasAttribute>  <# visible >,
     <#hasAttribute>  <# visible_class_B >
```

Code snippet 3-11: Inferred assertions of library attribute instance “Visible”

These reasoning inconsistencies regarding the class membership of individuals and the assignment of inconsistent default values led us to try and define an alternative mapping approaches regarding the ADOxx system elements.

3.1.4.2 ADOxx system elements as OWL 2 classes

The drawbacks of the first mapping approach led us to the second and the final one, as far as mapping of ADOxx system elements is concerned. According to this approach, every ADOxx system element is mapped to an OWL 2 class.

The Figure 3-17 diagrammatically represents the second approach, according to which every system attribute defined in the ADOxx global container is mapped to an OWL 2 class, and every instance of the attribute assigned to a core element is mapped to an ABox individual, which is a member of the corresponding OWL 2 class.

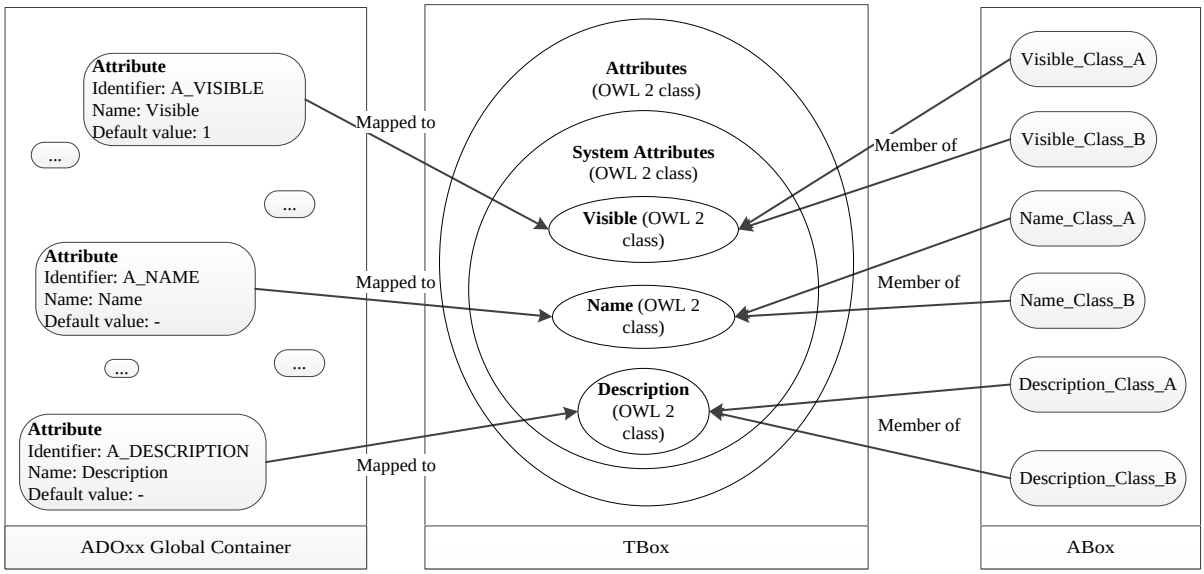


Figure 3-17: ADOxx system attributes – global container instances represented as OWL 2 classes and library instances as OWL 2 individuals

The OWL 2 classes represent the ADOxx system element defined in the global container. Every instance of the ADOxx system element owned by an ADOxx core element is now represented as an individual within ontology'ABox, and member of the corresponding OWL 2 class.

This approach allows us to express adequately the relationship between ADOxx system attributes and their assigned instances without inferential problems, regarding class memberships.

3.2 Lessons Learned

Mapping ADOxx meta-metamodel to ontology components has been a long and iterative, process. One of the very first conclusions that we came to, was that this is not a 1-to-1 process, meaning that there has never been only one correct approach, but always more than one, all of them equally expressive and close to ADOxx metamodelling platform philosophy. Factors that influenced our decision regarding which approach should be followed can be summarized as follows:

- ontology understandability,
- ontology maintainability,
- ontology extendibility,
- inferential consistency,
- reasoning performance and
- expressivity

Regarding the mapping of ADOxx *core elements*, our major findings can be summarised as follows:

- The hierarchical structure of the OWL 2 classes representing the ADOxx core elements has been proved to be a crucial task, as it provides a better understanding of the ontology, and the concepts represented within its TBox. Main advantages of its use are the reinforcement of ontology's understandability, and the ability to exploit the mechanisms that come with its correct use (property inheritance, etc.).
- The OWA as basis of OWL 2 makes the use of the "disjointness" concept almost imperative. It should always be taken into consideration when defining OWL 2 classes, for avoiding vague definition of classes and of their interrelations, which would potentially lead to inferential inconsistencies.

Regarding ADOxx *core relations*, identifying their type and their semantic properties contributed into their correct definition in OWL 2. ADOxx core relations expressing the concepts of "aggregation" or "association" were defined as OWL 2 object properties. ADOxx core relations expressing the notion of "generalization" were expressed within the OWL 2 class hierarchy.

As far as the definition of ADOxx *core element properties* is concerned, two factors have been taken into consideration during their definition in OWL 2: their Boolean nature, and the restrictions that ADOxx metamodelling platform imposes to their combination.

Mapping ADOxx core element properties to OWL 2 data properties has been proved to be an effective approach, until empirical study showed that the extended use of data properties within an ontology affects reasoning performance (section 6.3).

Mapping ADOxx core element properties to OWL 2 classes helped us overcome the drawbacks of the previous approach, and still define properties in an expressive and efficient

way. Major finding during the establishment of this approach was the effect of the OWA upon the "assignment" of properties to ADOxx core elements. More specifically, it has been proved that it has to be explicitly stated which properties are assigned to a core element and which not. An individual that represents an ADOxx core element, by not being a member of an OWL 2 class representing a property, does not imply the lack of this property from this individual. This assumption could lead to inferential gaps and inconsistencies.

Finally, it was not before the ontology reasoning, that implications stemming from the wrong use of the *SameIndividual* axiom became clear and made us abandon the first mapping approach, regarding ADOxx *system elements*. More specifically, the use of this axiom for individuals that do identify the same thing, but they do not share the same properties (e.g. default value) led to inferential inconsistencies regarding individual class membership. This led us to the second and last approach, according to which, instances of ADOxx system attributes within the ADOxx global container are mapped to OWL 2 classes, with members all instances of ADOxx system attributes representing ADOxx library instances.

4 Defining ADOxx Meta-metamodel Static Semantics

Within chapter 3, we described in detail the mapping approach defined and followed for mapping ADOxx meta-metamodel main concepts to ontology main components. Within this chapter, we are going to focus on the ADOxx meta-metamodel static semantics. More specifically, we are going to complete the mapping process by describing in detail the approach established for integrating the ADOxx meta-metamodel semantic constraints into the ontology.

This chapter is divided into two main sections. The first section focuses on the semantic constraints enforced by the ADOxx metamodelling platform and describes in detail the approach followed for their definition. The second section focuses on the ADOxx semantic constraints not enforced by the ADOxx metamodelling platform and describes once again the approach for their definition. In both cases, semantic constraint patterns specification is provided. Additionally, in both cases there is a separate reference to the mechanisms and their services used for validating these semantic constraints. The last section gathers all conclusions and lessons learned during this process.

4.1 Main Definition Approach

In contrast to the majority of the existing meta-metamodels, ADOxx meta-metamodel is tightly coupled with a concrete set of static semantics, which further constraint its syntax. ADOxx metamodelling platform provides an adequate mechanism for executing the needed static semantic checks and validating a metamodel against the semantic constraints imposed by the ADOxx meta-metamodel static semantics.

The integration of ADOxx semantic constraints into the ontology implies the use of two main elements: a language with a high expressive power for defining the semantic constraints and a mechanism for executing static semantic checks and identifying possible violation of the semantic constraints. The language that it is going to be used for the semantic constraints definition is the ontology language OWL 2. More specifically, the OWL 2 Manchester syntax is going to be used, which is a user-friendly compact syntax for OWL 2 ontologies. The mechanism for validating the semantic constraints against the semantic checks is the reasoning process provided by the currently available semantic reasoners.

A thorough look at the ADOxx meta-metamodel provides a clear picture of its syntax. An ADOxx specific example of a syntactic rule would be that only a relation class can have an endpoint. On the other hand, it is quite hard to identify semantic constraints defined within the ADOxx meta-metamodel. It is clear, for instance, that a relation class can have minimum two endpoints, but we get no information about the fact that a relation class *must* have exactly one “FROM” and exactly one “TO” endpoint.

Developing a semantic constraints definition approach has been a strenuous process, with many factors having to be taken into consideration:

- the nature of the semantic constraints,
- the way they are imposed, and
- the semantic reasoner services that can be used for identifying a potential violation

Main prerequisite for starting defining the ADOxx meta-metamodel semantic constraints was a deep analysis, during which we tried to identify and provide solutions to the previously listed issues. By the end of this analysis, it was quite clear that the ADOxx semantic constraints can be classified into two main categories: to those that are enforced by the ADOxx metamodelling platform and to those that are not (Figure 4-1).

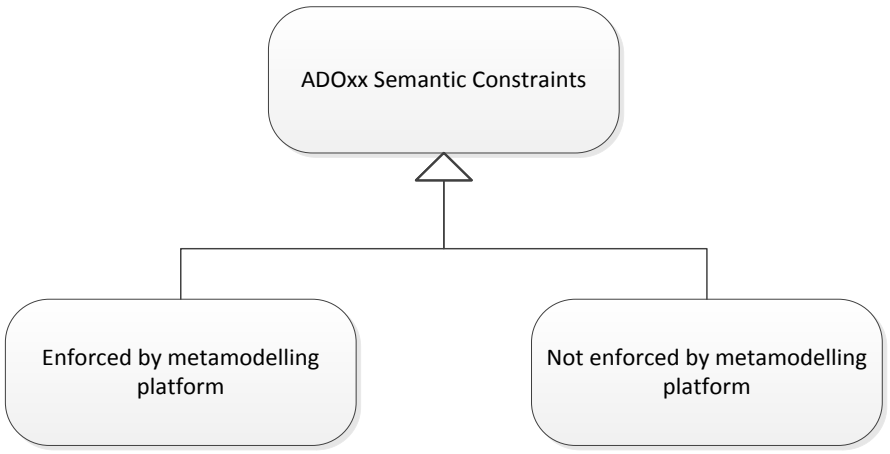


Figure 4-1: ADOxx semantic constraints classification

Throughout the following sections, we are going to describe the definition of these two types of ADOxx semantic constraints with the use of OWL 2. Additionally, the specified constraint patterns will be presented, as well as the way semantic reasoners services were used for detecting violations of the semantic constraints.

4.1.1 Semantic constraints enforced by metamodelling platform

It was shortly after having started establishing our mapping approach regarding the ADOxx meta-metamodel main concepts that we identified the first type of semantic constraints. This type of semantic constraints stemmed from the property assignment of the ADOxx main elements. Common characteristic of this type of constraints was that all of them were enforced by the metamodelling platform

A semantic constraint is considered to be enforced by the metamodelling platform, when it is imposed in such a way that there is no possibility to violate it. Thus, no semantic checks for identifying their violation are needed.

An ADOxx specific example of such a semantic constraint is that an ADOxx core class cannot be defined as *visible* and *abstract* simultaneously. This means that only one of the two properties can be assigned at once.

Having extensively explained and analysed the properties that can be assigned to every ADOxx core element in the section 2.1.3, this type of semantic constraints focuses on the forbidden combinations of the properties. Furthermore, two main patterns of such semantic constraints can be found within ADOxx meta-metamodel:

1. Properties that are mutually exclusive (disjoint sets) (Figure 4-2) and
2. Properties that the assignment of one imposes the assignment of another (one set is a subset of the other) (Figure 4-3).

Regarding the first pattern, an ADOxx specific example would be that an ADOxx core class instance cannot be defined at the same time as *abstract* and *visible,* because these two properties are considered to be mutually exclusive.

Getting into the field of set theory, Figure 4-2 represents the previous example regarding the first semantic constraint pattern in a graphical way. In such a case, within the set theory, the two properties would be represented as two disjoint sets, and every ADOxx core class instance as a dot. A dot that is a member of a set means that this property has been assigned to the ADOxx core class instance. The fact that the two sets are disjoint means that there cannot be an instance of an ADOxx core class that is a member of the two sets simultaneously.

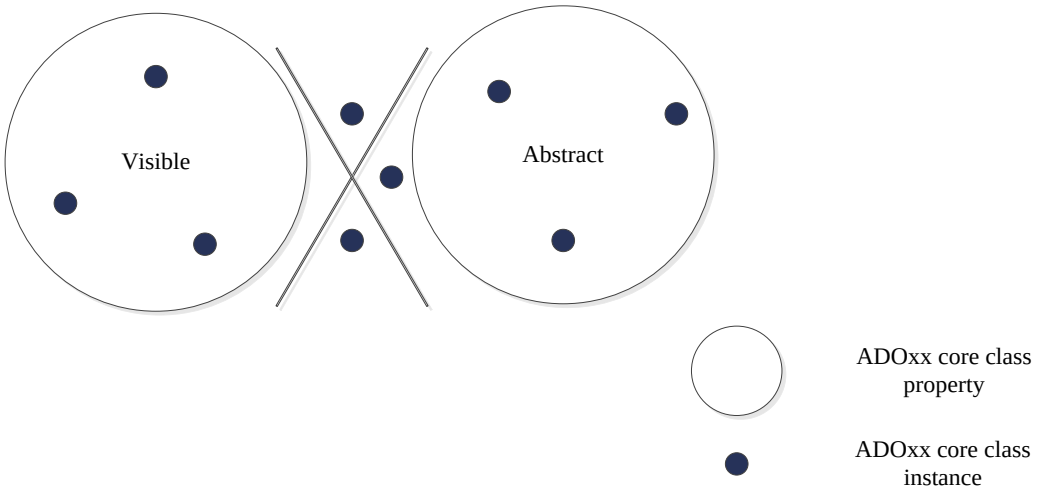


Figure 4-2: Example of mutually exclusive ADOxx core element properties

An example of the second semantic constraint pattern would be that an ADOxx relation instance cannot be defined as *interref*, without being defined as *repository*, as well. In this case, the assignment of the first property imposes the assignment of the second one too.

Based on the set theory, such a case would be represented as shown in the Figure 4-3. In this case, properties are again represented as sets, but this time the one set is a subset of the other. ADOxx relation instances are depicted as dots. A dot that is a member of a set means that this property has been assigned to the ADOxx relation instance. In this case, an ADOxx relation instance cannot be member of the subset *interref*, without being member of its super set *repository*.

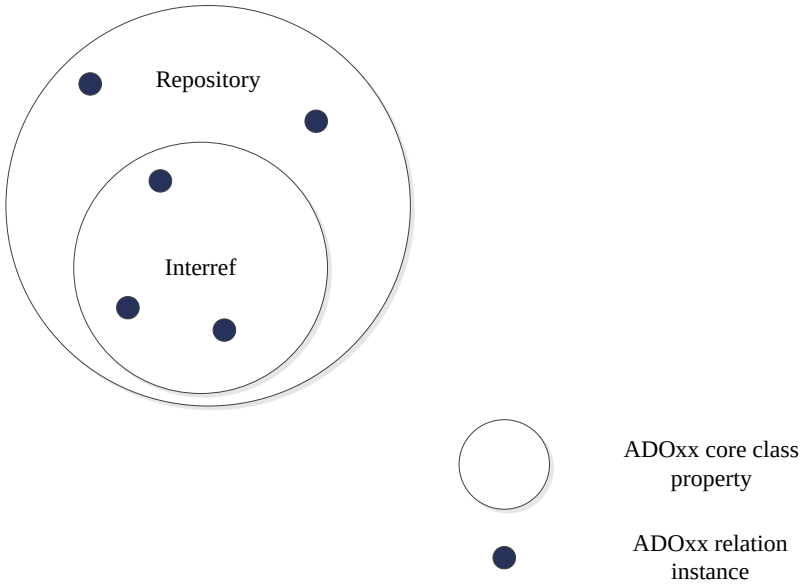


Figure 4-3: Example of ADOxx core element properties related with the relationship "sub-property"

According to the mapping approach defined in chapter 3.1.3, every ADOxx core element property is mapped to an OWL 2 class. Semantic constraints enforced by the metamodelling platform concern the property assignment to ADOxx core elements. Thus their definition with the OWL 2 ontology language requires the use of OWL 2 class axioms.

More specifically, the first semantic constraints pattern expresses the notion of mutual exclusiveness. The OWL 2 *DisjointClasses* class axiom expresses the notion of disjointness. Thus, semantic constraints that belong to this pattern make use of this class axiom. Regarding the second semantic constraints pattern, they express the notion of sub-property. The OWL 2 *SubClassOf* class axiom expresses the notion of the subset. Consequently, the definition of this type of semantic constraints requires the use of this class axiom.

Before we provide the semantic constraint patterns specification, a reference to the validation mechanism used for identifying their violation is necessary.

4.1.1.1 Inconsistency checking - reasoning service

Having identified the first type of semantic constraints, and having specified their definition with the OWL 2 ontology language, we should now try to identify the validation mechanism that could be used for identifying their violation.

According to section 2.2.6, validation of ontologies written in OWL 2 is based on semantic reasoners. Semantic reasoners provide a set of validation services (listed and briefly explained in section 2.2.6.1). Main objective of this task was to identify the appropriate reasoning service.

The way this type of semantic constraints was defined left no space for doubts regarding the reasoning service used for identifying their violation. ABox individuals representing instances of ADOxx main core elements that do not conform to the OWL 2 class axioms used for defining the semantic constraints, provoke failure of the *inconsistency checking* reasoning service. Consequently, failure of *inconsistency checking* reasoning service indicates violation of this type of semantic constraints.

The table below gathers the relevant cases regarding OWL 2 class axioms that cause failure of *inconsistency checking* service, and thus indicate semantic constraint violation.

Table 4-1: Cases of reasoning service *inconsistency checking* failure

Case	Trigger
DisjointClasses axiom violation	Definition of an ABox individual as member of two OWL 2 classes simultaneously that have been defined to be disjoint
SubClassOf axiom violation	Definition of an ABox individual as member of an OWL 2 class, but NOT as member of its OWL 2 super class

Inconsistency checking is one of the main core reasoning services. Failure of this service clearly indicates ontology incorrectness and is accompanied by an explanation providing information about the failure reasons.

A controversial issue regarding this reasoning service is how useful and self-descriptive the explanations are. Most of the times, explanations provide information that could be useful only for the constructor of the ontology. Thus, it is considered to be too "internal" and relevant only to the ontology constructor (Figure 4-4).

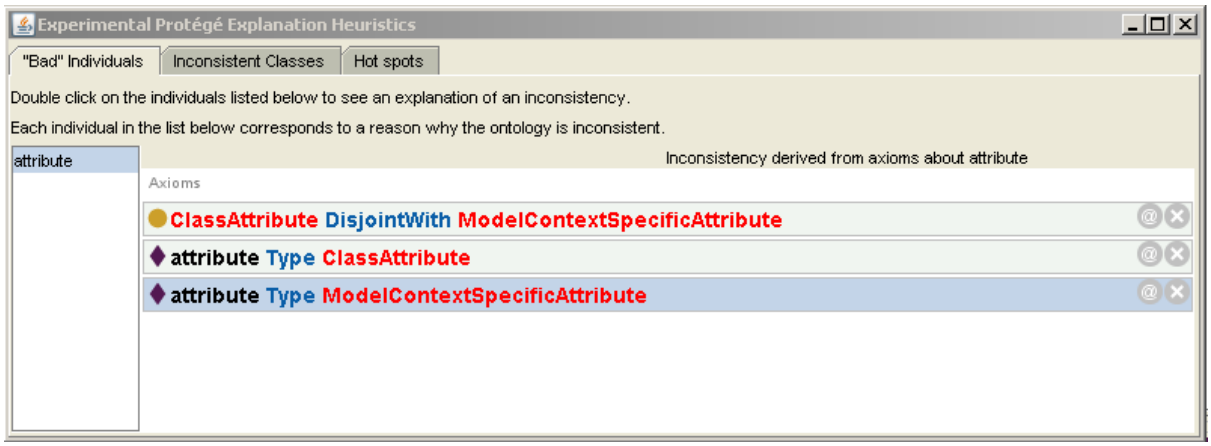


Figure 4-4: Explanation of failure of the reasoning service *Inconsistency checking*

Another issue is the fact that *inconsistency checking* is usually the first task semantic reasoners execute. Failure of this service interrupts the whole reasoning process. This, in addition to the lack of the ability to list all the reasons that led to the failure make the whole process of ontology correction quite slow.

After having completed the process of defining and validating this type of semantic constraints, within the next section, the semantic constraint patterns specification will be presented.

4.1.1.2 Constraint Patterns Specification

Specification of semantic constraint patterns regarding this type of semantic constraints was the outcome of our attempt to identify reoccurring semantic constraints. Main goal was to further classify them for simplifying their definition and making it simple and methodical with the use of the ontology language OWL 2.

Every pattern specification consists of the name of the pattern, the context of the semantic constraint, an example of the semantic constraint, the problem that rises from the semantic constraint and the solution provided to overcome the problem.

***Name*:** Mutually Exclusive Properties

Context: Two or more ADOxx core element properties are mutually exclusive. The assignment of one property to an ADOxx instance prevents the assignment of another property to the same one.

***Example*:** A *visible* ADOxx core class instance cannot be *abstract* at the same time.

Problem: How to identify the assignment of two or more mutually exclusive properties to an ADOxx core element instance.

Solution/ Implementation: For the definition of this type of semantic constraints the DisjointClasses OWL 2 axiom is used. ADOxx core element instances, represented as ABox individuals, that do not conform to the OWL 2 class axioms used provoke failure of the inconsistency checking reasoning service and indicate violation of this semantic constraint.

The Code snippet 4-1 represents the definition of two mutually exclusive properties in OWL 2, while the Code snippet 4-2 represents an individual definition that violates this semantic constraint.

```
Class: <#VisibleClass>

    SubClassOf:
        <#CoreClass>

    DisjointWith:
        <#AbstractClass>
```

Code snippet 4-1: *DisjointClass* axiom between *abstract* and *visible* properties

```
Individual: <#class_a>

    Types:
        <#VisibleClass>,
        <#CoreClass>,
        <#AbstractClass>,
        <#RepositoryClass>,
        not (<#TimeFilterRelevantClass>)
```

Code snippet 4-2: Violation of semantic constraint - *abstract* & *visible*

***Name*:** Sub-properties

Context: The assignment of one ADOxx core element property imposes the assignment of another one or more than one properties.

***Example*:** An interref ADOxx relation instance has to be repository too

Problem: How to identify the assignment of one ADOxx core element property to and ADOxx core element instance without the assignment of its super property.

Solution/ Implementation: For the definition of this type of semantic constraints the SubClassOf OWL 2 axiom is used. ADOxx core element instances, represented as ABox individuals, that do not conform to the OWL 2 class axioms used provoke failure of the inconsistency checking reasoning service and indicate violation of this semantic constraint.

The Code snippet 4-3 represents the definition of two properties related with the relation "sub-property" in OWL 2, while the Code snippet 4-4 represents the definition of an individual that violates this semantic constraint.

```
Class: <#InterrefRelationClass>

    SubClassOf:
        <#RepositoryRelationClass>
```

Code snippet 4-3: Class hierarchy between *repository* & *interref* OWL 2 classes

```
Individual: <#relation_a>

    Types:
        <#VisibleRelationClass>,
        <#RelationClass>,
        <#InterrefRelationClass>,
        not (<#RepositoryClass>)
```

Code snippet 4-4: Violation of semantic constraint - *time filter relevant* & *not repository*

4.1.2 Semantic constraints not enforced by metamodelling platform

The second main type of ADOxx semantic constraints concerns the constraints not enforced by the ADOxx metamodelling platform.

A semantic constraint is considered not to be enforced by the metamodelling platform when it is possible to be violated. Violation of such constraints is identified with the use of semantic checks.

An ADOxx specific example of such a semantic constraint would be that a model type definition must always contain the attribute definition "NAME". The attribute definition "NAME" can be missing from a model type definition, but this is semantically incorrect, and thus causes the failure of the corresponding semantic check.

A thorough study of ADOxx metamodelling platform semantic checks, along with an extensive use of ADOxx metamodelling platform for testing purposes led us to the conclusion that this type of constraints can be further divided into two main categories: those that are quantitative, and those that are qualitative.

The

Figure 4-5 depicts this classification of ADOxx semantic constraints not enforced by the metamodelling platform.

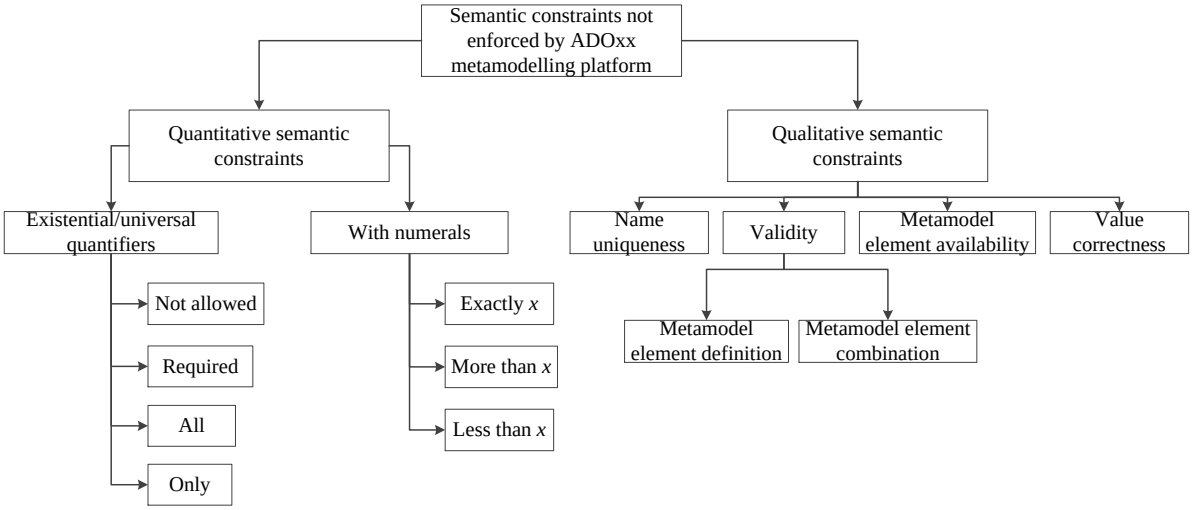


Figure 4-5: Classification of ADOxx semantic constraints not enforced by metamodelling platform

4.1.2.1 Quantitative Semantic Constraints

A semantic constraint is considered to be *quantitative* when it refers to relationships that the ADOxx core element instances participate in. A quantitative semantic constraint can be either existential/universal, or with numerals.

Existential/universal semantic constraints refer to constraints regarding:

- instances that are *not allowed* to participate in specific relationships
  - E.g.: The attribute definition has been found at the library object. This is not allowed. It must only be added to the class.
- instances that are *required* in specific relationships
  - E.g.: The library doesn't contain the system class 'REPOSITORY'. This class is required for each library.
- relationships that require *all instances of a specific ADOxx main concept*
  - E.g.: The model type for query results does not contain all visible classes and relation classes.
- relationships with *instances of only a specific ADOxx main concept*
  - E.g.: The model type contains only invisible classes/relation classes.

Quantitative semantic constraints with numerals refer to constraints upon a specific number of instances of an ADOxx main concept. More specifically, they restrict the number of instances that are allowed, required or forbidden to participate in specific relations. There are three main types of quantitative constraints with numerals:

- Exactly *x*
  - E.g.: The relation class needs exactly 2 endpoint definitions.
- more than *x*
  - E.g.: The library contains more than one model type for query results.
- less than *x*
  - The context definition for the model type has fewer parameters than the context definition for the repository.

4.1.2.2 Qualitative Semantic Constraints

A semantic constraint is considered to be qualitative when it refers to a qualitative property of an ADOxx main concept. A qualitative constraint can refer to:

- Name uniqueness
  - E.g.: The class definition doesn't have a unique language independent name within the model type.
- Validity
  - E.g.: The attribute of the model type should be a class attribute
  - E.g.: The parallel usage of the class 'USER_PROXY' and the class 'USER' in one library is not allowed.
- Metamodel element availability
  - E.g.: The endpoint definition of the relation class references the model type that is not contained in this library.
- Value correctness
  - E.g.: The endpoint cardinalities of the relation class are invalid. The maximum cardinality of the endpoint cannot be 0.

Validity refers to either metamodel element definition validity, or to the validity of the combination of metamodel elements.

In OWL 2, classes and property expressions are used to construct class expressions, also known as descriptions, which in the description logic literature, are called complex concepts. Class expressions represent sets of individuals by formally specifying conditions on their properties; individuals satisfying these conditions are said to be instances of the respective class (as described in section 1.2.2.1.3).

An OWL 2 class can either be *defined* or *primitive*. An OWL 2 class is considered to be *defined*, when there is an *EquivalenClass* axiom defined. On the other hand, an OWL 2 class is considered to be *primitive*, when it has either no class axiom, or only a *SubClassOf* axiom defined.

ADOxx semantic constraints due to their nature and their complexity could be considered as complex concepts, and consequently are represented as *defined* OWL 2 classes. ADOxx core element instances, represented as ABox individuals, that fulfil the conditions of the corresponding OWL 2 classes are inferred to be members of these complex classes. From now on, OWL 2 classes representing semantic constraints will be called constraint classes.

An example of a semantic constraint defined in OWL 2 as class expression is the following one (Code snippet 4-5):

```
Class: <#Con3Lib>

    Annotations:
        <#component> "Library"^^rdfs:Literal,
        <#checkType> "Error"^^rdfs:Literal,
        <#superClass> "Library"^^rdfs:Literal,
        rdfs:seeAlso "True"^^rdfs:Literal,
        <#identifier> "LILC 0001"^^rdfs:Literal,
        rdfs:label "Error: The library contains no language."^^rdfs:Literal

    EquivalentTo:
        <#Library>
         and (<#hasAvailableLanguage> some <#Language>)

    SubClassOf:
        <#Library>
```

Code snippet 4-5: ADOxx semantic constraint defined in OWL as class expression

Before presenting the specification of the semantic constraint patterns, the reasoning service used for identifying the violation of these constraints will be explained within the next section.

4.1.2.3 Classification - reasoning service

Every semantic constraint that belongs to this type is represented as a *defined* OWL 2 class. ADOxx core element instances, represented as individuals, that fulfil the conditions of the OWL 2 class are inferred to be members of this OWL 2 class – called constraint class.

In order to infer individual class membership, OWL 2 class axioms need to be carefully used. Class expressions within the *SubClassOf* axiom are considered to be necessary, but not sufficient conditions. This means that no class membership can be inferred after the completion of the reasoning process. On the contrary, conditions within *EquivalentClass* axiom are necessary and sufficient conditions, whose fulfilment infers class membership after the reasoning process.

OWL 2 provides a rich set of primitives that can be used to construct class expressions. In particular, it provides the well-known Boolean connectives *and*, *or*, and *not*; a restricted form of universal and existential quantification; number restrictions; enumeration of individuals; and a special *self*-restriction.

The Figure 4-6 represents which of the OWL 2 class axioms are considered to express just necessary conditions, and which of them express sufficient and necessary conditions.

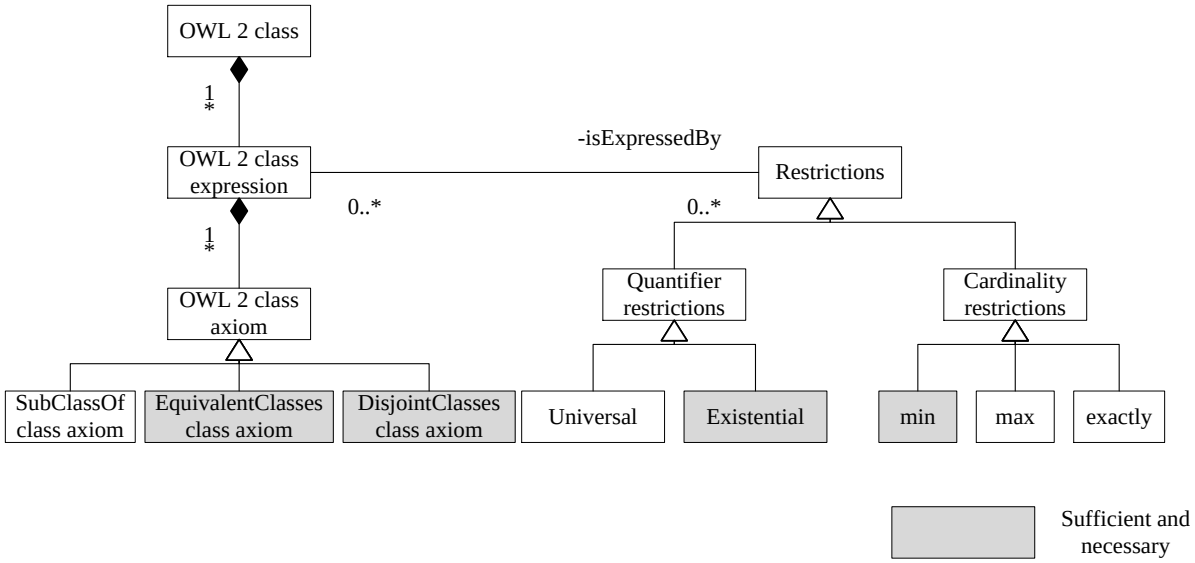


Figure 4-6: OWL 2 class expressions - necessary and sufficient and necessary axioms

The representation of semantic constraints as OWL 2 classes with conditions, whose fulfilment infers class membership, made the use of *classification* reasoning service for validating the constraints to seem the only logical choice. By classifying the ABox individuals, which represent ADOxx core element instances, against the TBox OWL 2 classes, which partly represent semantic constraints, we could automatically identify any violation of semantic constraints.

A reference regarding the representation of semantic constraints as OWL 2 classes is going to be made within the next section.

4.1.2.4 Representing semantic constraints as OWL 2 classes

An ADOxx semantic constraint not enforced by the metamodelling platform is tightly coupled with a rich set of information, like, for example, information regarding the type of the constraint, the ID of the constraint, etc. In addition, every semantic constraint is accompanied by a description for making the logical meaning of the constraint human-readable.

Representing a semantic constraint as an OWL 2 class is a process of two main steps:

1. Transformation of the semantic constraint human readable information into OWL 2 annotations, and assignment of them to the OWL 2 class
2. Transformation of the logical meaning of the semantic constraint into an OWL 2 class expression

The human readable information is distributed among a number of annotations as follows:

checkType: refers to the type of the message and can be either an error, warning, or information.

component: refers to the component that the specific semantic constraint belongs to. It can be Library Kernel, Notebook definitions, Generic views, Modelling editor, Modelling general, Relations control, Repository, or User core.

identifier: refers to the error code of the semantic constraint. First letters specify the component that the constraint belongs to and the number stands for the serial number of the component in the component.

label: stands for the message that describes the violation of the semantic constraint.

seeAlso: can be either true or false, depending on the instances that are classified as members of this OWL 2 class. In the case that the value is *true*, individuals that are in compliance with a specific constraint are classified under this class. In case the value is *false*, individuals that violate the constraint are classified under this class. This annotation is later used by the java validator for printing out the correct individuals accompanied by the "label" annotation.

superClass: specifies the direct super class of the constraint OWL 2 class. This annotation is used later by the java validator for printing out the correct individuals.

An example of a semantic constraint and its human readable information defined as annotations is represented in the following Code snippet 4-6.

```
Class: <#Con3Lib>

    Annotations:
        <#component> "Library"^^rdfs:Literal,
        <#checkType> "Error"^^rdfs:Literal,
        <#superClass> "Library"^^rdfs:Literal,
        rdfs:seeAlso "True"^^rdfs:Literal,
        <#identifier> "LILC 0001"^^rdfs:Literal,
        rdfs:label "Error: The library contains no language."^^rdfs:Literal
```

Code snippet 4-6: ADOxx semantic constraint human readable information as OWL 2 annotations

Having defined the transformation of semantic constraint human readable information into annotation labels, within the next section the definition of the logical meaning of the semantic constraint as OWL 2 class expression will be described by defining the specification of semantic constraint patterns.

4.1.2.5 Constraint Patterns Specification

Specification of constraint patterns was the outcome of our attempt to identify reoccurring constraint definitions so as to classify them and make future definition simple and methodical, and not intuitive and random.

For every subtype of semantic constraints not enforced by the metamodelling platform, a pattern was defined. Every pattern consists of a name, a description of the semantic constraint

context, an ADOxx specific example, a description of the problem that rises from the definition of the semantic constraint in OWL 2, and the solution provided.

The following semantic constraint patterns concern the quantitative semantic constraints.

***Name*:** Presence of *not allowed* instances

Context: Under this constraint pattern fall all ADOxx semantic constraints that check for ADOxx instances of a concept related with not allowed ADOxx individuals of another concept.

Problem: How to identify not allowed relationships between ADOxx instances.

Solution/ Implementation: The OWL 2 class expression that corresponds to this type of semantic constraints makes use of the existential restriction "some", which can be interpreted as "at least one". Members of this constraint class are all individuals which are related with a not allowed individual. Members of this constraint class violate the semantic constraint; thus *seeAlso* annotation is set as "False".

Example: The model type contains the system class 'REPOSITORY'. Model types cannot contain this class.

```
Class: <Con1MT>

    Annotations:
        <#component> "Library"^^rdfs:Literal,
        <#identifier> "LILC 0099"^^rdfs:Literal,
        <#checkType> "Error"^^rdfs:Literal,
        rdfs:label "Error: A model type cannot contain directly the system class
\"REPOSITORY\"."^^rdfs:Literal,
        <#superClass> "Con2MT"^^rdfs:Literal,
        rdfs:seeAlso "False"^^rdfs:Literal

    EquivalentTo:
        <#hasContainedClass> some <#REPOSITORY>

    SubClassOf:
        <#Con2MT>
```

Name: Absence of *required* instances

Context: Under this constraint pattern fall all ADOxx semantic constraints that check for missing relations among ADOxx instances of different concepts.

Problem: How to identify ADOxx instances that lack a relationship with specific instances.

Solution/ Implementation: The OWL 2 class expression that corresponds to this type of semantic constraints makes use of the existential restriction "some", which can be interpreted as "at least one". Members of this constraint class are all individuals that are related with the required instance. Members of this constraint class are all individuals that do not violate the ADOxx semantic constraint. Thus the value of the *seeAlso* annotation is set to "True".

Individuals that do not belong to this constraint class violate the semantic constraint. Consequently, by subtracting the members of the constraint class from an OWL 2 class that

contains all individuals of this concept, we get the remaining ones that violate the semantic constraint.

Example: A library must contain directly the system class "REPOSITORY".

```
Class: <#Con1Lib>

    Annotations:
        <#component> "Library"^^rdfs:Literal,
        <#superClass> "Library"^^rdfs:Literal,
        <#checkType> "Error"^^rdfs:Literal,
        rdfs:seeAlso "True"^^rdfs:Literal,
        rdfs:label   "Error:   A   library   must   contain   directly   the   system   class
\"REPOSITORY\"."^^rdfs:Literal,
        <#identifier> "LILC 0098"^^rdfs:Literal

    EquivalentTo:
        <#Library>
         and (<#hasDirectClass> some <#REPOSITORY>)

    SubClassOf:
        <#Library>
```

Name: Presence of *only* instances of a specific type

Context: Under this constraint pattern fall all ADOxx semantic constraints that check for ADOxx instances of a concept related with only instances of another specific concept. This can be either correct, or false.

Problem: How to identify ADOxx instances that are related with only instances of a specific concept.

Solution/ Implementation: The OWL 2 class expression that corresponds to this type of semantic constraints makes use of the existential restriction “some”, which can be interpreted as “at least one”. In case that an ADOxx instance must be related only with instances of a specific concept, the relation with an individual of another concept violates the semantic constraint. On the contrary, in case that an ADOxx instance shouldn’t be related only with instances of a specific concept, the relation with an individual of another concept would conform to the semantic constraint.
In the former case, the *seeAlso* annotation is set to “False“, as members of this constraint class violate the semantic constraint. In the latter case, the *seeAlso* annotation is set to “True“, as members of this constraint class do not violate the semantic constraint.

Example: A model type should not contain only *invisible* core or relation classes.

```
Class: <#Con4MT>

    Annotations:
        <#component> "Library"^^rdfs:Literal,
        <#identifier> "LILC 0019"^^rdfs:Literal,
        rdfs:seeAlso "True"^^rdfs:Literal,
        <#superClass> "Con2MT"^^rdfs:Literal,
        rdfs:label  "Warning:  A  model  type  should  not  contain  only  \"Invisible\"  core  or
relation classes."^^rdfs:Literal,
        <#checkType> "Warning"^^rdfs:Literal

    EquivalentTo:
        <#hasContainedClass> some
            (<#VisibleClass>
             or <#VisibleRelationClass>)
```

```
    SubClassOf:
        <#Con2MT>
```

Name: Presence of *all* instances of a specific type

Context: Under this constraint pattern fall all ADOxx semantic constraints that check for ADOxx instances of a concept that are related with all instances of another specific concept.

Problem: How to identify ADOxx instances that are not related with all instances of a specific concept.

Solution/ Implementation: Due to the OWA inference that an individual of a class is related with *all* individuals of another class is not straightforward. An alternative solution would be to check if all individuals of the second class are related with the specific individual of the other class.

This type of ADOxx semantic constraints is implemented with the use of two constraint classes, the one of which is a subclass of the other. Members of the sub-constraint class are only the individuals of a class that are related to a specific individual of another class. Members of the super constraint class are all individuals that should be related with a specific individual of another class.

The value of the *seeAlso* annotation is set to "true". The difference regarding the individuals between the two constraint classes are the individuals that violate the constraint. The *superClass* annotation must have the name of the super constraint class.

Example: The model type for query results does not contain all visible classes. The following visible core class is missing.

```
Class: <#Con17CoreClass>

    Annotations:
        <#component> "Library"^^rdfs:Literal,
        rdfs:seeAlso "True"^^rdfs:Literal,
        <#identifier> "LILC 0010"^^rdfs:Literal,
        rdfs:label "Warning: The model type for query results does not contain all visible
classes. The following visible core class is missing."^^rdfs:Literal,
        <#superClass> "Con18CoreClass"^^rdfs:Literal,
        <#checkType> "Warning"^^rdfs:Literal

    EquivalentTo:
        <#Con18CoreClass>
         and (<#classBelongsToMT> some <#QueryResultModelType>)

    SubClassOf:
        <#Con18CoreClass>

Class: <#Con18CoreClass>

    EquivalentTo:
        <#VisibleClass>
         and (<#classBelongsToLibrary> some
            (<#Library>
             and (<#hasModelType> some <#QueryResultModelType>)))

    SubClassOf:
        <#VisibleClass>
```

Name: *exactly* x instances

Context: Under this constraint pattern fall all ADOxx semantic constraints that check for ADOxx instances of a concept related with a specific number of instances of another specific concept.

Problem: How to identify ADOxx instances that are related with not an exact number of instances of a specific concept.

Solution/ Implementation: In this case, due to the OWA inference that an individual of a class is related with a specific number of individuals of another class is not straightforward. Thus, this type of ADOxx semantic constraints is implemented with the use of two constraint classes, the one of which is a subclass of the other. Members of the sub-constraint class are only the individuals of a class that are related to a number of individuals of another class that is greater than the allowed one. Members of the super constraint class are all individuals that are related with the allowed number of individuals of the other class.

Only members of the super constraint class do not violate the ADOxx semantic constraint, and the value of the "seeAlso" is set to "True. The "superClass" annotation must be filled with the class name that contains all individuals that should be related with the required individuals, so as to calculate the difference and get the remaining ones, that lack this relation. Members of the sub-constraint class are only individuals that violate the ADOxx semantic constraint, and more specifically because they are related with a greater number of individuals than the allowed one. The value of "seeAlso" annotation is set as "False".

Example: A relation class must contain exactly one "FROM" and "TO" endpoint.

```
Class: <#Con1Rel>

    Annotations:
        <#component> "Library"^^rdfs:Literal,
        <#superClass> "Con2Rel"^^rdfs:Literal,
        <#checkType> "Error"^^rdfs:Literal,
        <#identifier> "LILC 0030
"^^rdfs:Literal,
        rdfs:label "Error: A relation class cannot have more than one \"FROM\" or  \"TO\"
endpoint."^^rdfs:Literal,
        rdfs:seeAlso "False"^^rdfs:Literal

    EquivalentTo:
        <#hasEndPoint> min 3 <#EndPoint>

    SubClassOf:
        <#Con2Rel>

Class: <#Con2Rel>

    Annotations:
        <#component> "Modelling and Library"^^rdfs:Literal,
        <#checkType> "Error"^^rdfs:Literal,
        rdfs:seeAlso "True"^^rdfs:Literal,
        <#identifier> "MOLC 0016 LILC 0030"^^rdfs:Literal,
        <#superClass> "RelationClass"^^rdfs:Literal,
        rdfs:label "Error: A Relation Class must contain exactly one \"FROM\" and \"TO\"
endpoint."^^rdfs:Literal

    EquivalentTo:
        <#RelationClass>
         and (<#hasFromEndPoint> some <#EndPointFrom>)
         and (<#hasToEndPoint> some <#EndPointTo>)

    SubClassOf:
```

```
        <#RelationClass>
```

Name: *more than* x instances

Context: Under this constraint pattern fall all ADOxx semantic constraints that check for ADOxx instances of a concept that are related with a greater than the allowed number of instances of another concept.

Problem: How to identify ADOxx instances that are related with a greater than the allowed number of instances of a specific concept.

Solution/ Implementation: The OWL 2 class expression that corresponds to this type of semantic constraints makes use of a cardinality restriction, and more specifically of the *min* restriction. Members of this constraint class are all individuals of a class that are related with at least one more individual of the number allowed. The value of the "seeAlso" annotation is set to "False", because the member of this constraint class violates this type of semantic constraint.

Example: The library contains more than one model types for query results.

```
Class: <#Con13Lib>

    Annotations:
        <#component> "Library"^^rdfs:Literal,
        <#checkType> "Error"^^rdfs:Literal,
        <#identifier> "LILC 0009
"^^rdfs:Literal,
        <#superClass> "Con19Lib"^^rdfs:Literal,
        rdfs:label "Error: The library contains more than one model types for query
results."^^rdfs:Literal,
        rdfs:seeAlso "False"^^rdfs:Literal

    EquivalentTo:
        <#hasModelType> min 2 <#QueryResultModelType>

    SubClassOf:
        <#Con19Lib>
```

The implementation of ADOxx semantic constraints that refer to the type "less than *x*" could not be translated into OWL 2.

The following semantic constraint patterns concern the qualitative semantic constraints.

Name: Instance name uniqueness

Context: Under this constraint pattern fall all ADOxx semantic constraints that check for ADOxx elements that belong to a specific ADOxx concept and have the same name.

Problem: How to identify ADOxx instances that do not have a unique name.

Solution/ Implementation: OWL 2 does not allow two individuals to have the same name. This rule applies when individuals are manually created. However, in case than individuals are automatically imported (in our case with the mechanism that transforms an ADOxx metamodel into an ontology ABox) this restriction is not applied. Thus, the implementation of this type of ADOxx semantic constraints cannot be defined in OWL 2.

Example: The class definition doesn't have a unique language independent name within the model type.

Name: Instance incorrect value

Context: Under this constraint pattern fall all ADOxx semantic constraints that check for ADOxx instances with an invalid value.

Problem: How to identify ADOxx instances with an invalid value assigned.

Solution/ Implementation: This type of ADOxx semantic constraint could be implemented with the use of facets. Facets are used for constraining the range of data properties. However, the big number of attributes whose value should be checked, in addition to our decision not to make use of data properties, didn't make the implementation of this type of semantic constraints possible.

Example: The endpoint cardinalities of the relation class are invalid. The maximum cardinality of the endpoint cannot be 0.

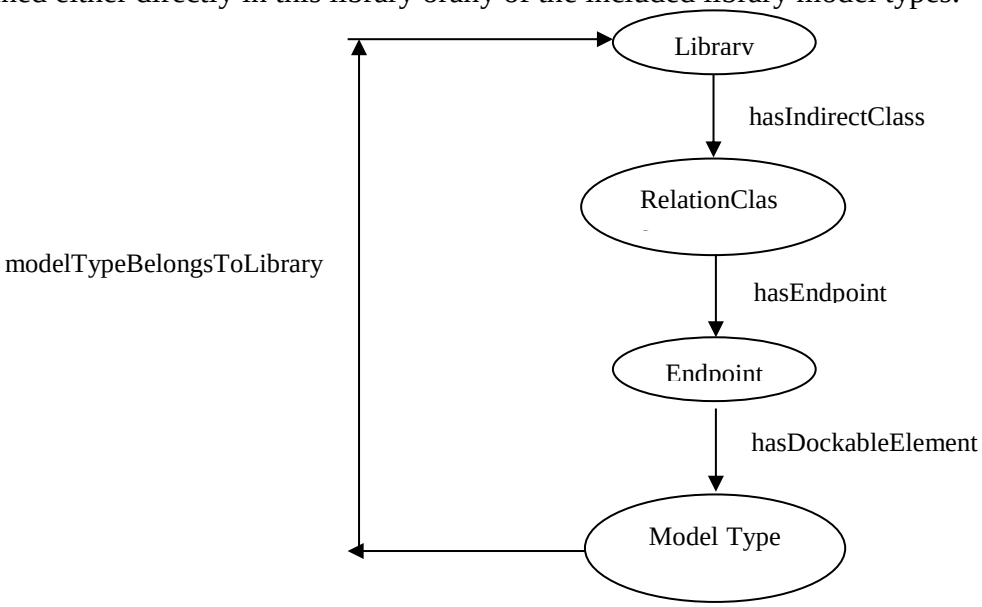
Name: Metamodel instance availability

Context: Under this constraint pattern fall all ADOxx semantic constraints that check for ADOxx instance availability within the metamodel.

Problem: How to identify ADOxx instances that are not available within the metamodel.

Solution/ Implementation: This type of ADOxx semantic constraints could be implemented with the use of SWRLs. SWRLs, with the use of built-in axioms can express cyclicity of concepts. Their computational complexity though, had a great effect on the reasoning process, and thus this type of constraints was not implemented.

Example: The endpoint definition of the relation class references the class that is not contained either directly in this library orany of the included library model types.



Name: Incorrect instance definition

Context: Under this constraint pattern fall all ADOxx semantic constraints that check for the correct definition of ADOxx instances within metamodel.

Problem: How to identify incorrect ADOxx instance definitions.

Solution/ Implementation: This type of ADOxx semantic constraint is implemented by defining a new constraint class, members of which are all individuals whose definition is incorrect. The value of the "seeAlso" annotation is set to "False".

Example: The attribute definition of the endpoint definition at the relation class is not model context specific. Attribute definitions of endpoint definitions for modelling relation classes also have to be model context specific.

```
Class: <#Con14Attr>

    Annotations:
        <#component> "Library"^^rdfs:Literal,
        <#identifier> "LILC 0084"^^rdfs:Literal,
        <#checkType> "Error"^^rdfs:Literal,
        rdfs:label "Error: The attribute definition of the endpoint definition at the relation
class is not model context specific. Attribute definitions of endpoint definitions for
modeling relation classes also have to be model context specific."^^rdfs:Literal,
        rdfs:seeAlso "False"^^rdfs:Literal,
        <#superClass> "Attribute"^^rdfs:Literal

    EquivalentTo:
        (<#Attribute>
         and (not (<#ModelContextSpecificAttribute>)))
         and (<#attrBelongsTo> some
            (<#EndPoint>
             and (<#endPointBelongsTo> some <#ModellingRelationClass>)))

    SubClassOf:
        <#Attribute>
```

Name: Incorrect instance combination

Context: Under this constraint pattern fall all ADOxx semantic constraints that check for ADOxx elements' correct definition within metamodel.

Problem: How to identify incorrect ADOxx instance combinations.

Solution/ Implementation: This type of ADOxx semantic constraint is implemented by defining a new constraint class, members of which are all individuals who are related with a not allowed individual. Thus the value of the "seeAlso" annotation is set to "False".

Example: The parallel usage of the class 'USER_PROXY' and the class 'USER' in one library is not allowed.

```
Class: <#Con6Lib>

    Annotations:
        <#component> "Repository"^^rdfs:Literal,
        <#identifier> "RELC 0032"^^rdfs:Literal,
        <#superClass> "Library"^^rdfs:Literal,
        <#checkType> "Error"^^rdfs:Literal,
        rdfs:label "Error: The parallel usage of the class 'USER_PROXY' and the class 'USER'
in one library is not allowed."^^rdfs:Literal,
        rdfs:seeAlso "False"^^rdfs:Literal

    EquivalentTo:
        <#Library>
         and (((<#hasDirectClass> some <#USER>)
         and (<#hasDirectClass> some <#USER_PROXY>))
         or ((<#hasIndirectClass> some <#USER>)
         and (<#hasIndirectClass> some <#USER_PROXY>)))

    SubClassOf:
        <#Library>
```

4.2 Lessons Learned

Within this chapter, we extensively described the process of integrating ADOxx semantic constraints into the ontology representing the ADOxx meta-metamodel. The language used for the definition of ADOxx semantic constraints was the ontology language OWL 2, whereas the validation mechanism used for identifying semantic constraint violations was semantic reasoning.

The main factors that had to be taken into consideration during the definition process can be summarized as follows:

- The big number of ADOxx semantic constraints
- The logical nature and the complexity of ADOxx semantic constraints
- The expressive power of OWL 2
- The restrictions that stem from the use of OWL 2
- The services provided by semantic reasoning for identifying violations of semantic constraints

Main objectives that influenced in a great extend the decisions made regarding the approach followed for the definition of ADOxx semantic constraints were:

- The correct interpretation of ADOxx semantic constraints and their precise translation into OWL 2
- The maximum coverage of ADOxx semantic constraints translated into OWL 2
- The assurance of the consistency and the correctness of the semantic constraint validation
- The efficiency of the semantic constraint validation mechanism

Classification of ADOxx semantic constraints was the outcome of our attempt to deal with their big number and to identify reoccurring constraints, so as to classify them and make future definition simple and methodical, and not intuitive and random. This contributed in a great extend in moderating the drawback regarding their big number. ADOxx semantic constraints were classified into two main categories:

- ADOxx semantic constraints *enforced* by the metamodelling platform: they are imposed in such a way that there is no possibility to violate it. Thus, no semantic checks for identifying their violation are needed.
- ADOxx semantic constraints *not enforced* by the metamodelling platform: it is possible to be violated. Violation of such semantic constraints is identified with the use of semantic checks.

The first type of semantic constraints focuses on the forbidden combinations of ADOxx core element properties. They are further classified into those that concern mutually exclusive properties, and those that concern sub-properties. The second type of semantic constraints

was further classified into those that are considered to be quantitative, and those that are qualitative.

The high complexity of ADOxx semantic constraints, together with the high expressive power of OWL 2, imposed a continuous tendency to translate the semantic constraints into complex OWL 2 class expressions. This had a great effect on the performance of the validation mechanism, and more specifically resulted in a time consuming validation process. Keeping OWL 2 class expressions as clean, simple and concrete as possible was the main lesson learned during the ADOxx semantic constraints translation into OWL 2 class expressions.

Main obstacles stemming from the OWL 2 that had to be overcome can be summarized into the two following points:

- The concept of the OWA
- The lack of cyclic dependencies support

The bottom layer of the logical architecture of the ADOxx metamodelling platform, as shown in the Figure 1-1 consists of persistency services and more specifically of a Database Management System (DBMS). Persistency services support the durable storage of models and metamodels. These services abstract from concrete storage techniques and permit storing of modelling information in heterogeneous data sources such as files, databases or web services. The use of a DBMS implies the appliance of the CWA (section 2.2.3.2) and the necessity of negation as failure (NaF). This contradicts the basis of OWL 2 which is the OWA. Closing the open world locally (e.g. selectively use of *DifferentIndividuals* axiom) did partly contribute to overcome this drawback. However, ADOxx semantic constraints regarding metamodel element availability, or lack of required instances could not be translated into OWL 2.

Regarding *cyclic dependency of concepts*, non-existence of such dependencies is considered to be a major prerequisite for maintaining decidability, and computational estimation reasonable of an ontology. OWL 2 does not provide us with the ability to define cyclic class dependencies. This drawback could be overcome by enriching OWL 2 expressive power with the use of SWRLs. Their computational complexity though, and the effect that it had across reasoning performance led us to the decision not to implement this type of semantic constraints.

The semantic reasoning services used for identifying ADOxx semantic constraint violations were tightly dependent on their translation into OWL 2. Violation of constraints under the former category is detected by causing failure of ontology *inconsistency checking*. Violation of constraints that belong to the latter category is detected with the use of *classification* reasoning service. The use of the latter reasoning service necessitated the use of OWL 2 axioms that are considered to be necessary and sufficient, so as class membership to be inferred after the reasoning process.

5 Building an ADOxx Semantic Constraint Validation Mechanism

ADOxx metamodelling platform provides a mechanism for executing semantic checks upon semantic constraints of ADOxx meta-metamodel. Within the previous chapters, a holistic approach was defined for transforming the ADOxx meta-metamodel and its metamodels into an ontology, including the ADOxx semantic constraints. In this chapter, we are going to describe the prototype validation mechanism developed for implementing the ADOxx semantic checks for checking and identifying violation of semantic constraints.

The implementation of the prototype is supported by the current available various semantic technology implementations, APIs, tools which deal with annotation handling, OWL 2 and semantic reasoners. Main motivation for the implementation of this prototype was our attempt to take full advantage of what semantic technologies have to offer, so as to build an efficient validation mechanism for implementing the ADOxx semantic checks and identifying violation of ADOxx semantic constraints using the current available reasoning mechanisms of the existing reasoners combined with the services concerning annotation handling that OWL API provides.

5.1 Choosing Software Development Model

Before starting designing and implementing the prototype, the first step was to define the software development model that would be followed. After an extensive literature research, we found a number of software development models, with the most dominant ones to be the Waterfall Model, the V Model, the Spiral Model and the RAD Model.

After taking into consideration our requirements, and evaluating the advantages and disadvantages of the existing software development models, it became quite clear that an iterative software development model should be followed. The relatively small size of the project and the fact that, either new requirements, or corrections could occur anytime on the fly led us to the decision that the model that we should follow should provide us with the ability to iterate to a previous phase whenever was needed. Thus, we decided to follow an iterative software development model.

Iterative development (Cockburn 2008) is a rework scheduling strategy in which time is set aside to revise and improve parts of the system. Requirements and user interfaces are the most common causes of revising the current work.

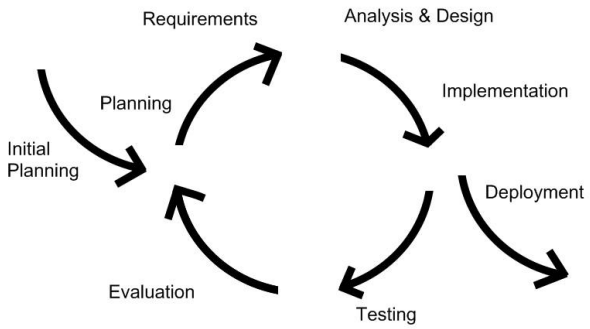


Figure 5-1: Iterative development

The guidelines that define the iterative software development are the following:

• In case of any difficulty in designing, coding and testing a modification, redesigning and recoding is necessary so as for the problem to be solved.

• Modifications should be implemented to the modules easily. If that is not possible, then redesigning is in order.

• As the project progresses, the modifications should become easier.

• The existing implementation should be frequently analysed to see if it matches to the required criteria of the software to be delivered.

5.2 Defining Requirements

Major goal was to develop a mechanism that would reason the given ontology, annotated with semantic knowledge, and would generate a report regarding all individuals that violate a semantic constraint. For this purpose, the existing semantic web technologies, and the already proven highly optimized semantic reasoners are going to be used. The prototype developed should fulfil the following requirements ((BSSC) 1995):

Functional requirements:

- Validation of a given ADOxx metamodel against the ADOxx semantic constraints.
- *Given Input*: the ontology representing the ADOxx meta-metamodel and its metamodel, annotated with semantic knowledge, and including the ADOxx semantic constraints.
- *Expected output*: a report regarding individuals that violate ADOxx semantic constraints, accompanied by the proper human-readable information.

Performance requirements:

- Regarding validation time, the whole validation process ideally should not take longer than the validation process that takes place within ADOxx metamodelling platform. In case that the first objective is not achievable, a time limit of 2 minutes should be set.

- Acceptable time: 1 minute
- Nominal time: 2 minutes
- Ideal time: less than 1 second

- Correctness of results is considered to be achieved, when the report of violated ADOxx semantic constraints is identical to that of ADOxx metamodelling platform semantic checks.

Interface requirements:

- Exploit annotations for reasoning purposes.
- Manipulate annotations for delivering the appropriate human-readable information regarding the reasons of ADOxx semantic constraint violations.
- Explanation of inconsistencies and incoherencies so that the user can apply ontology correction moves.

5.3 System Description

In this section, we are going to present and describe the architecture of the prototype validation mechanism and the way that it works. For the implementation and realization of the system, Eclipse IDE was used. The validation mechanism is implemented in Java programming language making use of the OWL API andthe libraries of the used reasoners.

OWL API is a high level Java Application Programming Interface for working with OWL 2 ontologies. The OWL API is closely aligned with the OWL 2 structural specification. It supports parsing and rendering in the syntaxes defined in the W3C specification; manipulation of ontological structures; and the use of reasoning engines. The reference implementation of the OWL API, written in Java, includes validators for the various OWL 2 profiles (Horridge and Bechhofer, The OWL API: A Java API for Working with 2009).

The design of the OWL API is directly based on the OWL 2 structural specification, as depicted in the figure below (Figure 5-2):

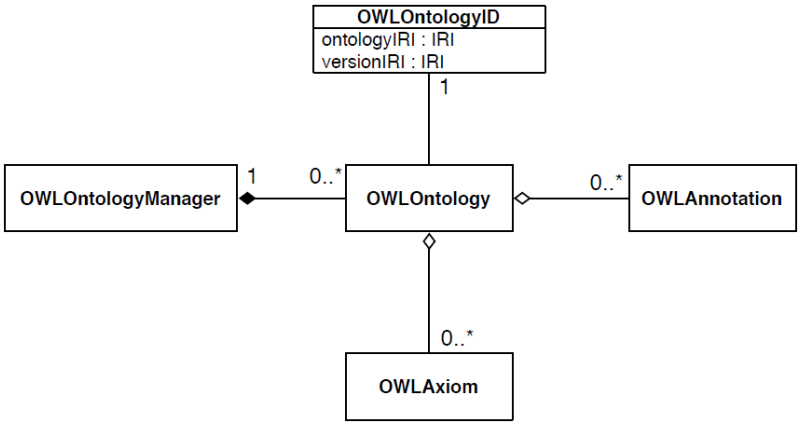


Figure 5-2: OWL 2 structural specification

5.3.1 Architecture

Within the architecture of the validation mechanism, three distinct layers can be distinguished: input, processing and output. In this section, we will discuss the technical aspect of each layer and its components with the use of algorithms or examples. It is to be noted though that algorithms are mainly used for giving an insight to understanding the prototype model developed.

The Figure 5-3 illustrates the three distinguished layers of our architecture.

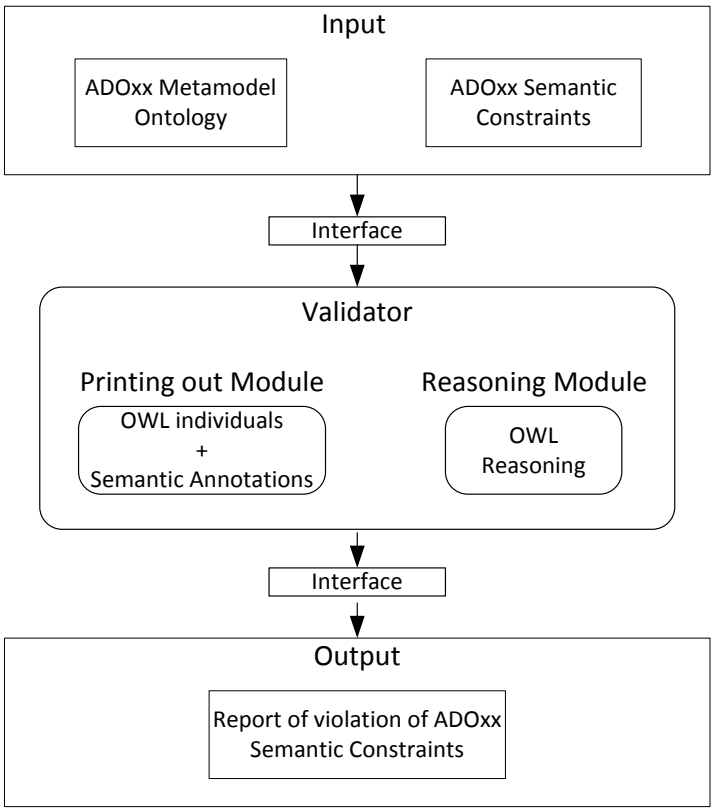


Figure 5-3: Architecture layers of the prototype validation mechanism

5.3.2 User Interface

As we can see in Figure 5-3, interfaces are provided before and after the processing module of the validation mechanism. The interfaces are used for two main reasons:

1. Allow the user to select the semantic reasoner that will perform ontology reasoning
2. Provide the user with a report, regarding the violated ADOxx semantic constraints

The Figure 5-4 represents the interface before the processing module for selecting the semantic reasoner to perform the ontology reasoning.

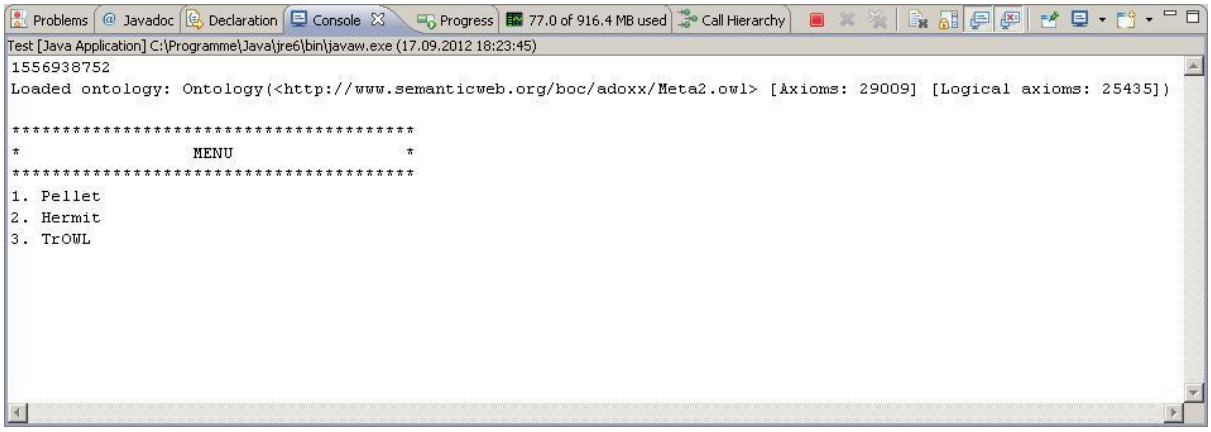


Figure 5-4: Semantic constraint validation mechanism interface

5.3.3 Input

Main input of our validation mechanism is the domain ontology. Domain ontology contains all information needed to produce the expected output. The information contained in the domain ontology could be decomposed into two sub-inputs.

- *ADOxx (meta)-metamodel*: the annotated domain ontology, to which all main concepts and relations of ADOxx meta-metamodel, as well as the ADOxx metamodels, are mapped.
- *ADOxx semantic constraints***:** they are integrated into the domain ontology represented as OWL 2 classes with an *EquivalentClass* axiom and accompanied by semantic annotations.

5.3.4 Functionality

The validation mechanism consists of two main processing modules: OWL 2 ontology reasoning and printing out the individuals violating semantic constraints, accompanied by the proper semantic annotation. After input is loaded, validation mechanism processes the ontology by realizing OWL 2 reasoning. After the completion of the reasoning process, all individuals are classified among the constraint classes. The second processing module prints out all individual classified among the constraint classes, together with human-readable information explaining the semantic constraint violation reasons.

The Figure 5-5 represents the way that the validation mechanism works.

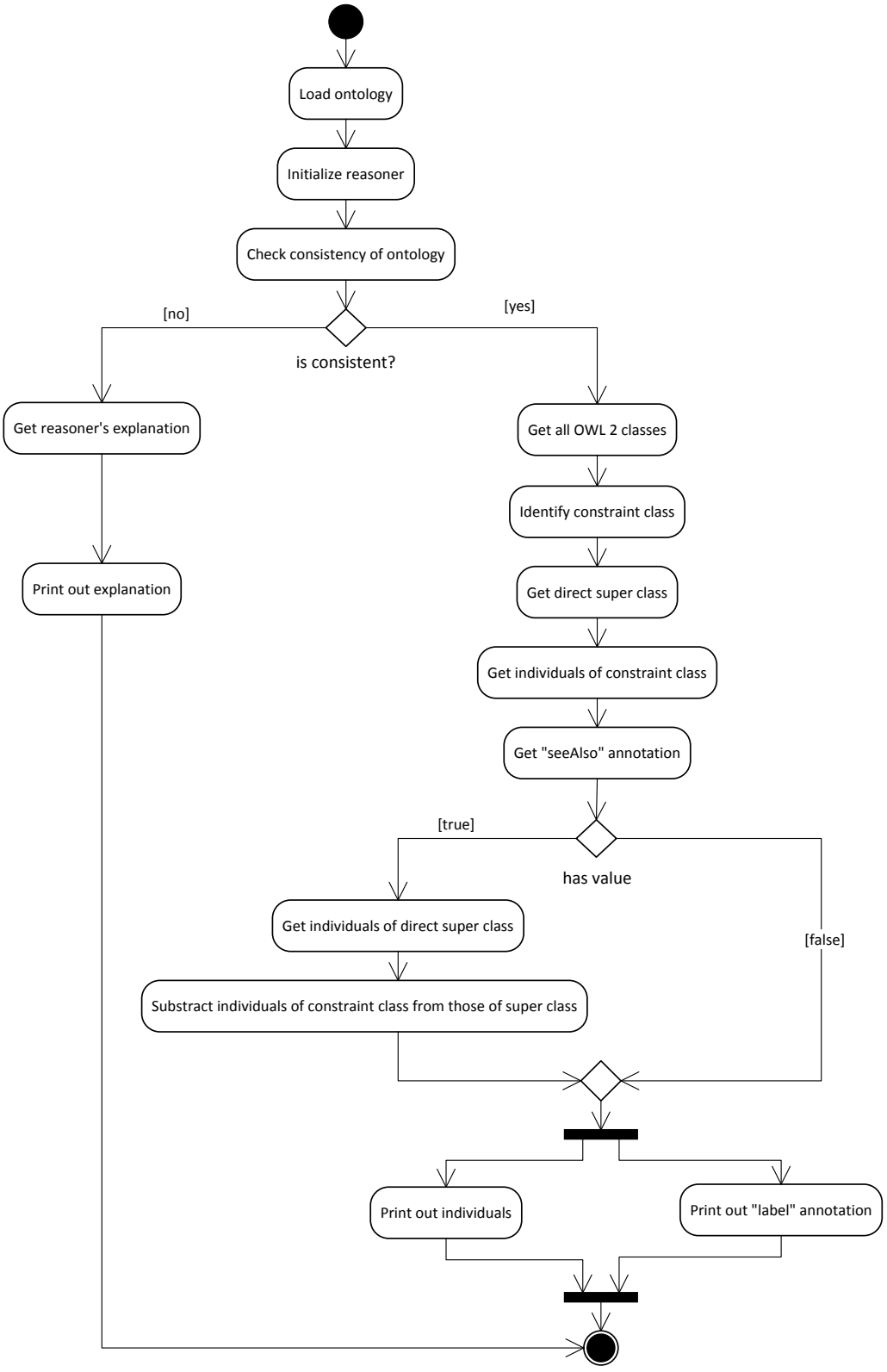


Figure 5-5: Validation mechanism functionality

5.3.4.1 Validation mechanism - OWL reasoning module

Ontology reasoning by a semantic reasoner serves two main purposes: to ensure ontology's consistency and to classify individuals among classes. An inconsistent ontology entails at least one individual that violates an ADOxx semantic constraint enforced by the metamodelling platform. Individuals that violate not enforced by the metamodelling platform ADOxx semantic constraints are classified among the corresponding constraint classes. Completion of the reasoning process provides a classified set of individuals among the constraint classes.

This module performs all reasoning tasks, with the use of the selected reasoner (Pellet, HermiT or TrOWL). It performs one of the most important tasks; it checks the consistency and coherency of the domain ontology passed as input, and it classifies the individuals among the constraint classes. All reasoners provide a direct interface to check the consistency of ontology and get the explanation of a potential inconsistency.

In case that the domain ontology is consistent, the execution of the printing out module follows. Otherwise, the reasoning process is interrupted, and the reason of inconsistency must be corrected in order to go on with the printing out module.

The Algorithm 5-1 shows the implementation to check ontology's consistency.

```
Input: domain ontology Ontology
Output: result of inconsistency checking
Initialize reasoner;
boolean var: isConsistent (Ontology)
If (!var)
       getExplanation():
else
       call printing_out_function();
end
```

Algorithm 5-1: Algorithm for checking ontology's consistency

5.3.4.2 Validation mechanism – printing-out module

In case that the domain ontology is proved to be consistent, the printing-out module is called. Main task of this module is to print out all individuals classified among the constraint classes that violate an ADOxx semantic constraint, together with the proper semantic annotation.

Depending on the *EquivalentClass* axiom, individuals that are members of these classes can either be the ones that violate an ADOxx semantic constraint, or individuals that do not violate a constraint. In the former case, these individuals together with the proper explanation are directly printed out, but in the latter case, individuals that do not violate the specific constraint have to be subtracted from the set that contains all those individuals that should fulfil this constraint and then print out the remaining ones, again with the proper explanation.

The Algorithm 5-2 shows the implementation of printing out the individuals and the proper explanation.

```
Input: domain ontology Ontology
Input: set of OWL classes σ, every one of each corresponds to an ADOxx metamodel
Core Element
Output: all individuals violating an ADOxx semantic constraint with the proper
explanation represented as a semantic annotation

Initialize reasoner;
Foreach Class ∈ σ
       Get direct subclasses;
       if (name of subclass begins with "Con")
              Get set of individuals;
              Get "seeAlso" annotation;
              if ("seeAlso" annotation is false)
                     foreach individual
                            print out individual;
                            print out "label" annotation;
                     end
              else
                     Get individuals of direct super class;
                     Subtract individuals of subclass from individuals  of
                     super class;
                     if (subclass contains individuals)
                            foreach individual
                                   print out individual;
                                   print out "label" annotation;
                            end
                     end
              end
       end
end
```

Algorithm 5-2: Algorithm for printing out individuals that violate semantic constraints with their explanation

5.3.5 Output

After processing the domain ontologym which is provided as input to the validation mechanism, a report containing all individuals that violate an ADOxx semantic constraint with a proper description, depending on the specific constraint that they do violate is provided as output. For this purpose, Eclipse IDE console is used.

Example 5-1: Part of ADOxx semantic constraint violations report

```
1 Error: The relation class is not marked as an interref. SOURCE:
<RC_PROVIDED_FUNCTIONS_CS_{4B0FD345-C1BC-4FF6-80A3-69A7143DC58A}>

9 Information: The class has no notebook definition. SOURCE: <REPOSITORY_{560E9F75-0059-4720-
8397-3E9A0BD65130}>

22 Warning: The relation class  is not marked as a visible class. SOURCE:
<TASK_ASSIGNEDTO_USER_{38D701BB-D24D-496F-AF5F-A1947FDDDC89}>

122 Information: The Relation Class doesn't contain the Attribute "Notebook". SOURCE:
<RC_DB_OWNER_{97492F94-6D10-4346-B8C9-6C907018846F}>

144 Error: A Modelling NOT interref Relation can connect:
- A Repository Core Class with a Modelling Core Class.
- A Repository Core Class with an other Repository Core Class. SOURCE: <IS_INSIDE_{2CB8C626-
2273-452C-BD3D-6EFC73310569}>
```

6 Evaluation of Reasoners Performance

After having described the theoretical basis and the background of our approach quite extensively within the previous chapters, the next step would be to study and test its applicability.

To demonstrate our approach and study its applicability in terms of performance and effectiveness, we had to validate real ADOxx metamodels and not simple and small-size ones that were created for testing purposes. Due to their size and complexity, it would be quite time consuming to transform manually the selected metamodels into ontologies. Thus, we implemented an ontology generator, which we called "Owlizer", in order to transform ADOxx metamodels into OWL 2 ontologies fast and effective. Main task of the "Owlizer" was to identify ADOxx metamodel elements and assign them to ontology's TBox main concepts accompanied by their properties and their interrelations.

With this means, we transformed three ADOxx metamodels into OWL 2 ontologies for testing purposes.

The Figure 6-1 provides some metrics concerning the size of ADOxx metamodels and thus the size of the OWL 2 ontologies corresponding to these metamodels.

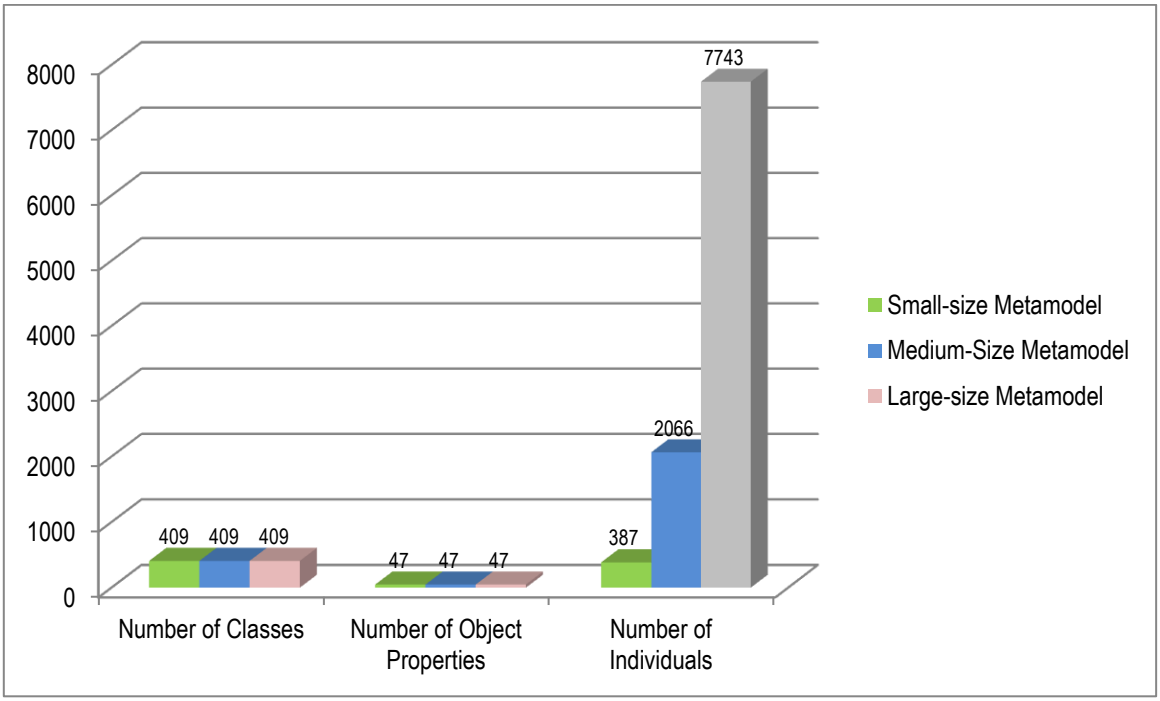


Figure 6-1: Size of ADOxx metamodels corresponding to the number of OWL 2 individuals

At the bar chart above there are three sets, each of which consists of three differently coloured bars. The first two sets refer to ontology's TBox, which represents ADOxx meta-metamodel and semantic constraints. Thus, in all three ontologies the number of OWL 2 classes and OWL 2 object properties is the same. Among OWL 2 classes, apart from ADOxx

core elements, we have ADOxx system elements, ADOxx core element properties and ADOxx semantic constraints. The third set corresponds to ontology's ABox, which represents the ADOxx metamodel, and is the one that differentiates the ontologies and classifies them according to their size. Within our approach, our small, medium and large-size ontologies consist of 387, 2.066 and 7.743 instances respectively.

At this point should be pointed out that comparison of reasoners performance took place at different phases of the approach definition andthe construction of the ontology. Thus, not all comparisons were performed with the same or the final version of the ontology. Ontologies used may vary regarding the number of ADOxx concepts defined, the number of ADOxx semantic constraints, as well as the size of the ABox. Whenever a prior or a semi-final version of the ontology is used for performing the comparison tests, the metrics of the ontology are provided.

6.1 Choosing Semantic Reasoner

Having fully transformed ADOxx metamodels into OWL 2 ontologies with the least possible loss of information, we had the input of the validator described in the previous section. Next step for transforming ontology's implicit knowledge into explicit and detecting violations of ADOxx semantic constraints was to determine the semantic reasoner that would be used within validator's reasoning module for reasoning the OWL 2 ontologies.

The choice was not that obvious, taking into consideration the amount of semantic reasoners that are currently used for this purpose. Main criteria for choosing the semantic reasoner to be used were firstly, its compatibility with the chosen OWL 2 sub-language, which in our case was OWL 2 DL and secondly its compatibility with OWL API. The semantic reasoners that fulfilled these two criteria and provided the best support as far as OWL 2 DL reasoning is concerned were the three following ones:

- Pellet reasoner
- HermiT reasoner
- TrOWL reasoner

After an extensive literature research trying to point out the strengths and weaknesses of these three semantic reasoners so as to select the one to be used in our approach, we came to the conclusion that, although all of them fulfilled the previously mentioned prerequisites, not all of them provide us with the same functionality. Pellet reasoner, for instance, provides full support of SWRL rules (Hitzler and Parsia 2009), in contrast to HermiT or TrOWL reasoners that do not. On the other hand, HermiT reasoner provides a better support of OWL 2 data property facets, in comparison to the other two semantic reasoners. These factors led us to the decision to use all three of them, trying to take advantage of their strengths according to our needs, and study their performance and capability of dealing with different sizes of OWL 2 ontologies.

During the definition of our mapping approach regarding the construction of ADOxx OWL 2 ontology, we established many alternative solutions and implementations. It was, among many factors, semantic reasoners capability of dealing with the alternative scenarios, as well as their performance; that influenced our decision concerning which implementation should be followed. Within the following sections we are going to present and describe these implementation scenarios and how semantic reasoners dealt with.

6.2 Ontology Size and Reasoning

The size of ADOxx metamodels can vary from very small ones to really large ones with a big number of class, relation, and attribute instances. Taking this into consideration, one of the main objectives was to find out how ontology's size affects reasoning performance. More specifically, we tried to study reasoners capability to deal effectively, not only with small-size, but also with large-size metamodels, and thus ontologies. For this purpose, we chose and

transformed three ADOxx metamodels of different sizes into three ontologies, and we reasoned all three of them with all three reasoners. The Figure 6-2 represents the time that each reasoner needed for reasoning the small, medium and the large-size ontology.

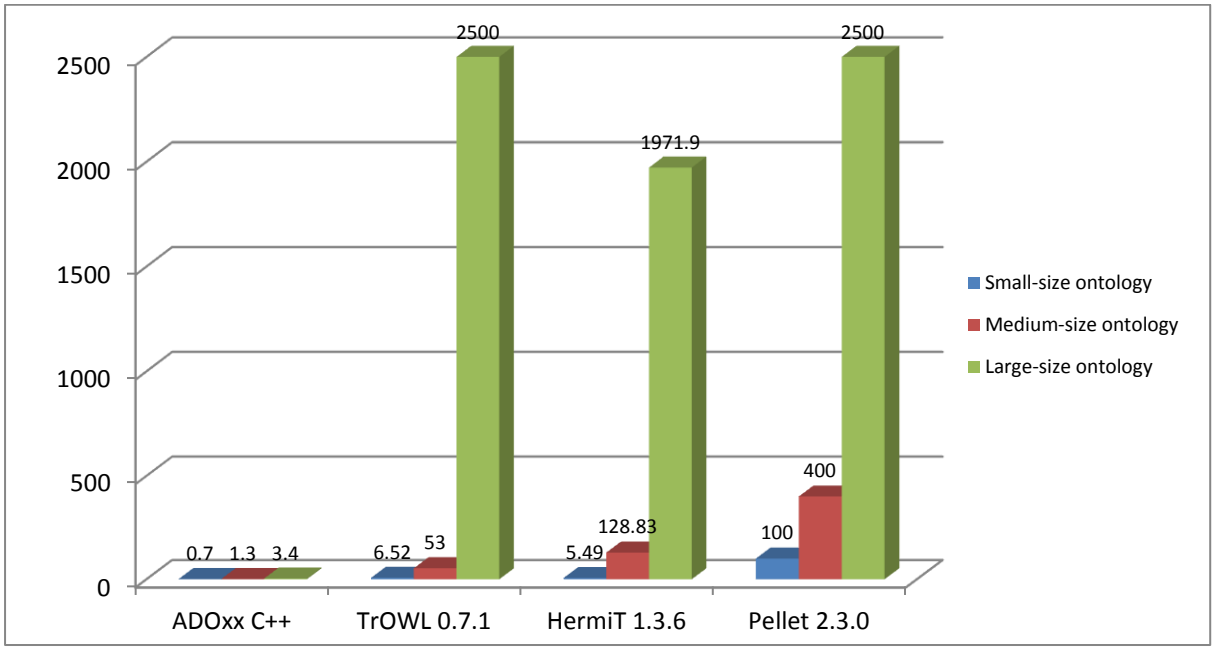


Figure 6-2: Correlation of reasoner performance with ontology size

The bar chart above represents the time measured in seconds that every semantic reasoner needed to reason all three different ontologies of different sizes. Thus, we have four sets of bars, with the first one referring to the validation mechanism of ADOxx metamodelling platform implemented in C++, the second one to the TrOWL 0.7.1 reasoner, the third one to the HermiT 1.3.6 reasoner and the fourth one to the Pellet 2.3.0 reasoner. Every set consists of three differently coloured bars. The blue bar represents the small-size ontology, the red bar the middle-size ontology and the green one represents the large-size ontology. The y-axis represents the time that every validation mechanism needed to validate/reason the metamodels measured in seconds. Our decision to include the time that validation mechanism of ADOxx metamodelling platform needs to validate various sizes of metamodels stems from the fact that this performance time is our ideal value that we defined during the requirements definition of validator's implementation, as far as performance requirements are concerned.

According to the bar chart, the ADOxx metamodelling platform succeeds the fastest validation of metamodels, independently from their size. HermiT reasoner succeeds the fastest reasoning of the small-size ontology among the semantic reasoners, with the TrOWL reasoner coming second and the Pellet reasoner third. Regarding the medium-size ontology, TrOWL reasoner achieves the fastest reasoning, with HermiT at the second place, and Pellet at the third. HermiT reasoner was the only semantic reasoner that managed to reason the large-size ontology within the time limit of 2.500 seconds that we have set. Neither TrOWL, nor Pellet managed to complete ontology's reasoning within this time limit. A possible reason could be their inability to deal effectively with a large number of interrelations within the ontology, stemming from the growing ABox of the ontology.

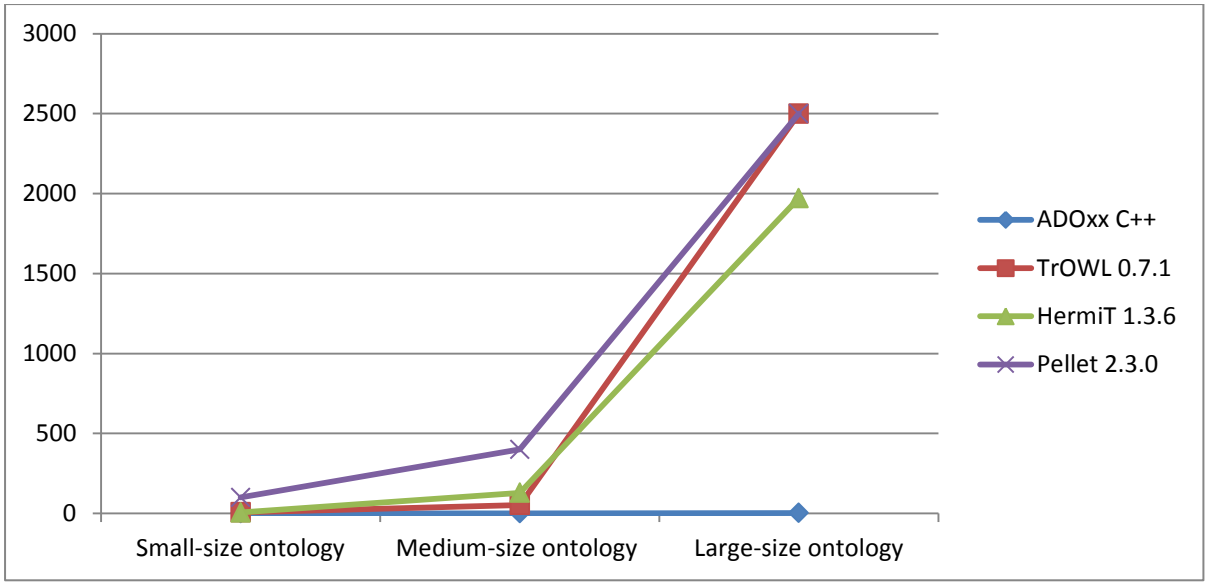


Figure 6-3: Trendline of reasoning performance based on ontology size

Judging from the Figure 6-3, we could come to a safe conclusion that reasoning time increases exponentially as the size of the ontology increases, and more specifically as the number of instances of ontology's ABox is getting approximately four times bigger than that of the smaller one.

6.3 OWL 2 Data Properties and Reasoning

During the definition of the mapping process regarding the ADOxx meta-metamodel to an ontology, we faced a dilemma, whether the ADOxx core element properties should be defined as OWL 2 classes or data properties. Since both implementations seemed to be in compliance with the ADOxx metamodel customization philosophy, it was reasoning performance which helped us come to a decision regarding which of these two approaches should be applied.

The two figures below (Figure 6-4, Figure 6-5) represent the reasoning performance of every semantic reasoner for both these two alternative implementations taking into consideration the size of the ADOxx metamodel. For this purpose, the small-size metamodel had to be left out, due to the fact that no conclusion could be drawn about how data properties affect reasoning performance, as the reasoning time was too small, and the influence could not be noticeable.

The metrics of the ontology used for the comparison of these approaches are gathered in the Table 6-1.

Table 6-1: OWL 2 data properties & reasoning - ontology metrics

	Ontology with data properties	**Ontology without data properties**
OWL 2 classes	164	164
OWL 2 object properties	29	29
OWL 2 data properties	**0**	**76**
OWL 2 individuals	643	643
OWL 2 SubClassOf axioms	159	159
OWL 2 EquivalentClasses	80	80
OWL 2 expressivity	SROIQ (D)	SROIQ

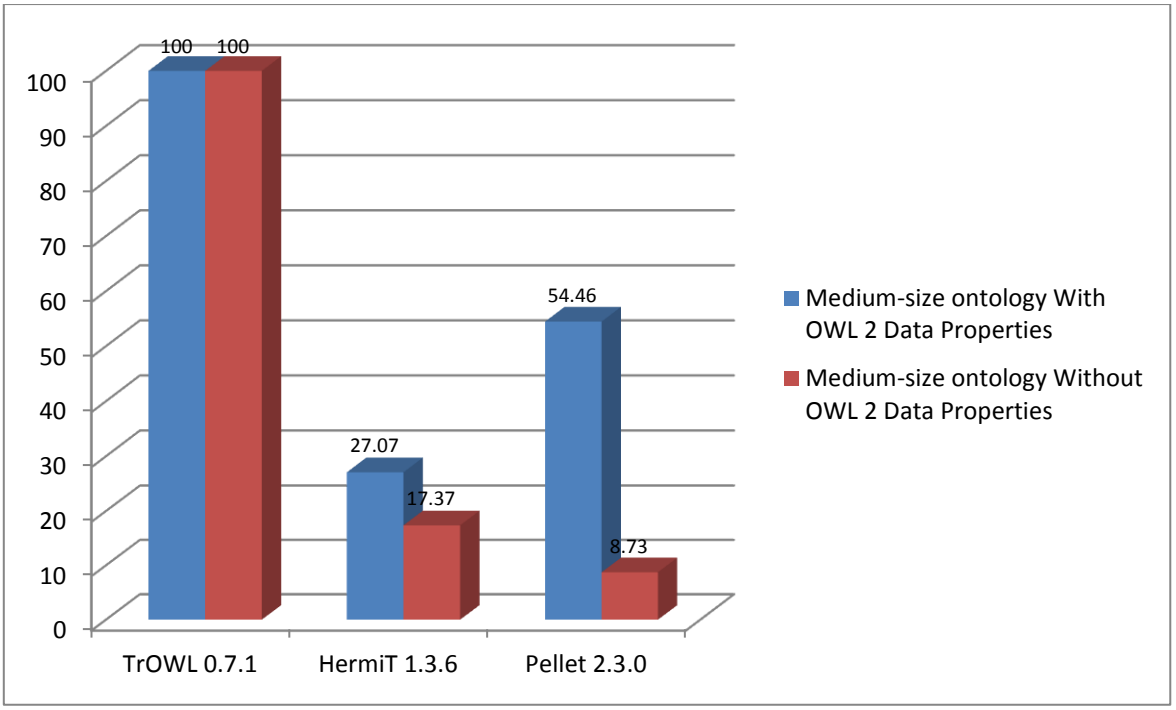


Figure 6-4: Correlation of OWL 2 data properties and reasoners performance - medium-size ontology

The bar chart of Figure 6-4 represents the effect of OWL 2 data properties upon the reasoning performance of all three semantic reasoners of the medium-size ontology. Thus, there are three sets of bars one for every semantic reasoner. The first set corresponds to the TrOWL 0.7.1 reasoner, the second set to the HermiT 1.3.6 and the third to the Pellet 2.3.0 reasoner. Every set consists of two differently coloured bars. The blue one represents the ontology where OWL 2 data properties were used for the implementation of ADOxx core element properties. The red one represents the second alternative approach, where the ADOxx core element properties are mapped to OWL 2 classes.

As shown in the bar chart above, HermiT reasoner seems to handle quite effectively both of the approaches judging from the ontology's reasoning time, which regardless the applied approach is the lowest compared to the other two reasoners. However, the use of OWL 2 data properties seems to slow down reasoning performance like the blue bar reveals. Pellet reasoner comes to the second place, with its reasoning performance being two times more than that of HermiT reasoner, regardless of the applied mapping approach. Likewise to the first case, the use of OWL 2 data properties seems to have a great effect on Pellet's reasoning performance by slowing it down approximately six times. Finally, TrOWL reasoner did not manage to complete ontology's reasoning within the time limit that was set, regardless the applied approach.

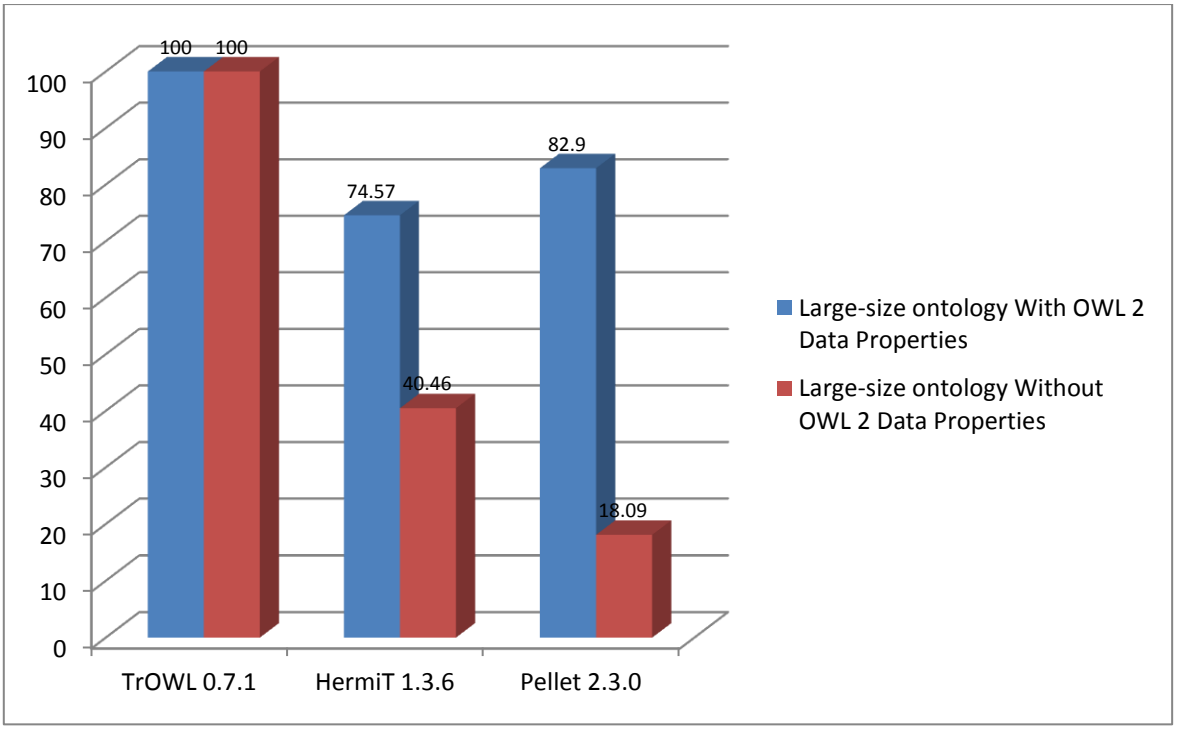


Figure 6-5: Correlation of OWL 2 data properties and reasoners performance - large-size ontology

The bar chart of Figure 6-5 concerns the large-size OWL 2 ontology and Figure 6-4 represents the effect of OWL 2 data properties upon the reasoning performance of all three semantic reasoners of the ontology. Likewise, there are three sets of bars one for every semantic reasoner. The first set corresponds to the TrOWL 0.7.1 reasoner, the second one to the HermiT 1.3.6 and the third set to the Pellet 2.3.0 reasoner. Every set consists of two differently coloured bars each of which corresponds to the applied approach. The blue one represents the ontology where OWL 2 data properties were used for the implementation of ADOxx core element properties. The red one represents the second alternative approach, where the ADOxx core element properties are mapped to OWL 2 classes.

In the case of the large – size OWL 2 ontology, HermiT reasoner seems to handle quite efficiently the OWL 2 data properties, in comparison to the other two reasoners. On the other hand, Pellet reasoner succeeds a better reasoning time in case of the implementation of ADOxx core element properties as OWL 2 classes. Both reasoners though succeed a better reasoning time without the use of OWL 2 data properties, with the HermiT being approximately two times faster, and the Pellet about six times faster. TrOWL reasoner did not manage to complete ontology's reasoning within the time limit that was set, regardless the applied approach.

Regardless of the size of the ADOxx metamodel and consequently of the OWL 2 ontology, we could come to the conclusion that the use of OWL 2 data properties for the implementation of ADOxx core element properties do have an effect on reasoning performance. In case of the HermiT reasoner, reasoner performance seems to be two times faster without the use of OWL 2 data properties. Regarding Pellet reasoner, the

implementation of ADOxx core element properties as OWL 2 classes seems to be reasoned approximately five times faster in comparison to the first approach with the use of OWL 2 data properties.

6.4 Rules and Reasoning

While trying to categorize ADOxx semantic constraints and to define an approach for their implementation, we came across with one type of semantic constraints that due to its cyclic nature could not be implemented, but only with the use of DL safe rules.

DL safe rules and more specifically, their built-in axioms that allowed us expressing conceptual cyclicity was what actually led us to experiment with their use for the implementation of this type of semantic constraints.

An extensive literature research showed that not all semantic reasoners do support DL safe rules, and in case they do, either their integration is on early stage, or not all built-in functions are supported. In our case, as the experimentation with the use of rules started at an early stage of the definition of our approach, only Pellet 2.3.0 semantic reasoner was used for testing how the use of rules affects reasoning performance.

It was at an early stage of defining our approach, when ontology's size was not the final one, considering that by that time not all concepts, their instances, as well as all semantic constrains have been define, that we started applying DL safe rules. Their use, although it contributed in overcoming the issue regarding cyclic semantic constraints, it had a main disadvantage that became quite obvious from the beginning. The use of only one rule slowed down at a great degree reasoning performance. Taking this into consideration, in addition to the fact that more than one rule had to be defined for the implementation of this type of ADOxx semantic constraints, made us reconsider their use.

The metrics of the ontology regarding the comparison of these approaches are gathered in the Table 6-2.

Table 6-2: Rules & reasoning - ontology metrics

	Ontology metrics
OWL 2 classes	164
OWL 2 object properties	29
OWL 2 data properties	0
OWL 2 individuals	643
OWL 2 SubClassOf axioms	159
OWL 2 EquivalentClasses	80
OWL 2 expressivity	SROIQ

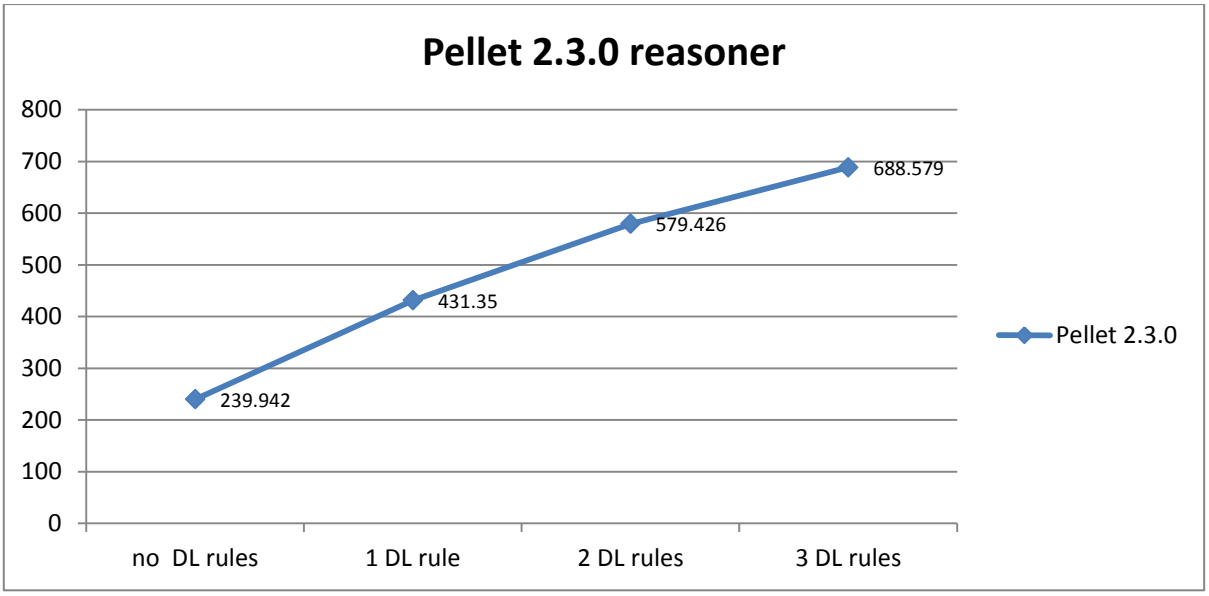


Figure 6-6: Correlation of DL safe rules and reasoners performance

The graph of Figure 6-6 represents the effect of using DL safe rules upon reasoning performance. The trendline seems to be upwards, meaning that reasoning performance was slowed down by approximately 150 milliseconds for every rule that was introduced.

As expected, at the last stage of our approach definition and of ontology's construction, when not only the majority of ADOxx concepts and their instances have been defined, but also most of the ADOxx semantic constraints, none of the semantic reasoners managed to complete reasoning with the use of rules, thus we could not have any comparison results.

6.5 Globally vs. Locally Closing the Open World and Reasoning

During our approach definition for defining semantic constraints of ADOxx meta-metamodel, we confronted a type of semantic constraints which required the explicit statement that instances of a specific concept are different among each other.

This type of ADOxx semantic constraints belongs to the constraint pattern *"more than "x" not allowed"*, where "x" is a specific number, and concerns mainly the restriction of the number of instances of a concept related via a relation with instances of another concept. An ADOxx specific example of such a semantic constraint would be that a relation cannot have more than one "FROM" endpoints.

As it is already extensively mentioned, OWL 2 is based on the OWA which fundamental principle is that everything is true, unless it can be proven false.

With that taken into consideration, two or more instances of a concept are by default not considered to be different, unless it is explicitly stated that these instances are different from each other. OWL 2 provides us with the ability of defining instances of concepts as different, a fact that contributes in avoiding misconceptions or not clear definition of instances.

Consequently, we came up with two alternative approaches for defining this type of ADOxx semantic constraints. The first one could be considered as an attempt to globally close the open world, by defining all instances of ontology's ABox as different from each other, in contrary to the second one that could be considered as an attempt to locally close the open world by defining as different only the ABox instances that participate in this type of semantic constraints.

In correspondence with the previous cases, we tried to compare reasoning performance for both the approaches, with the use of all three semantic reasoners and for all three sizes of ADOxx metamodels.

The figures below (Figure 6-7, Figure 6-8) represent the reasoning performance of every semantic reasoner for both these two alternative approaches of two of the three ontologies.

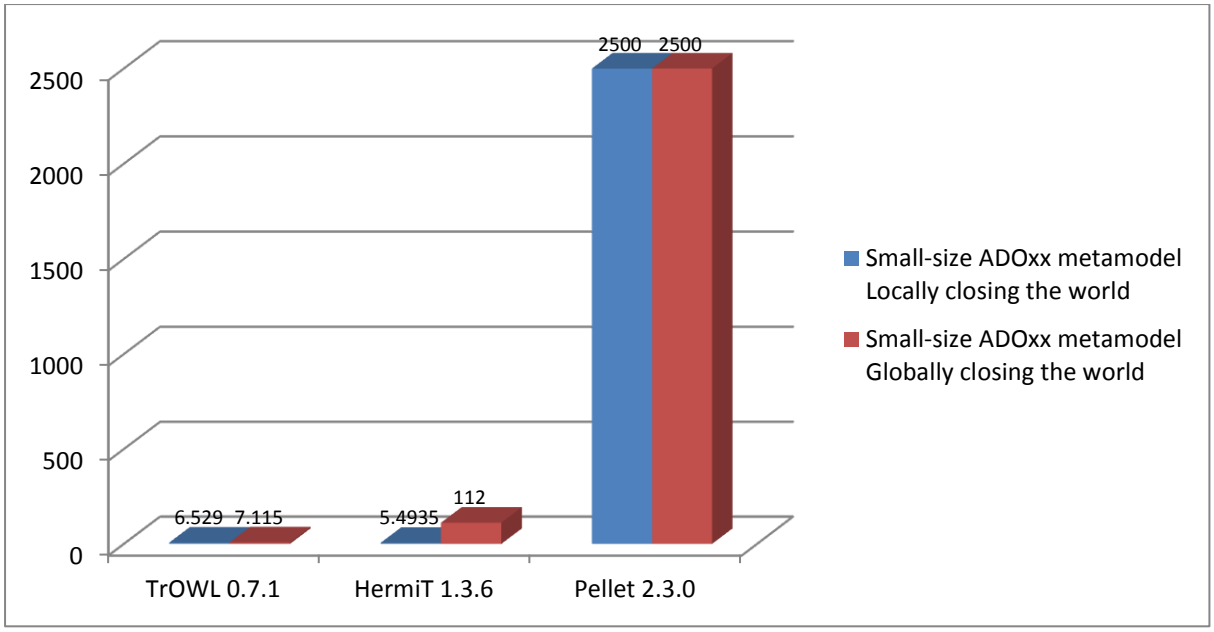


Figure 6-7: Correlation of globally/locally closing the world and reasoners performance - small-size ontology

The bar chart above concerns the small-size OWL 2 ontology and Figure 6-4 represents the effect of locally closing the world, in comparison to globally closing the world, upon the reasoning performance of all three semantic reasoners of the ontology. There are three sets of bars one for every semantic reasoner. The first set corresponds to the TrOWL 0.7.1 reasoner, the second one to the HermiT 1.3.6 and the third set to the Pellet 2.3.0 reasoner. Every set consists of two differently coloured bars each of which corresponds to the applied approach. The blue one represents the ontology where the open world has been locally closed. The red one represents the second alternative approach, where the open world has been globally closed.

TrOWL reasoner seems to handle both approaches quite effectively, as there is no difference upon the reasoning time. The second approach though seems to have a great effect on HermiT's performance, which seems to be twenty times slower in comparison to the first one, where the open world is locally closed. Pellet reasoner finally, didn't manage to complete reasoning of the ontology regardless of the approach.

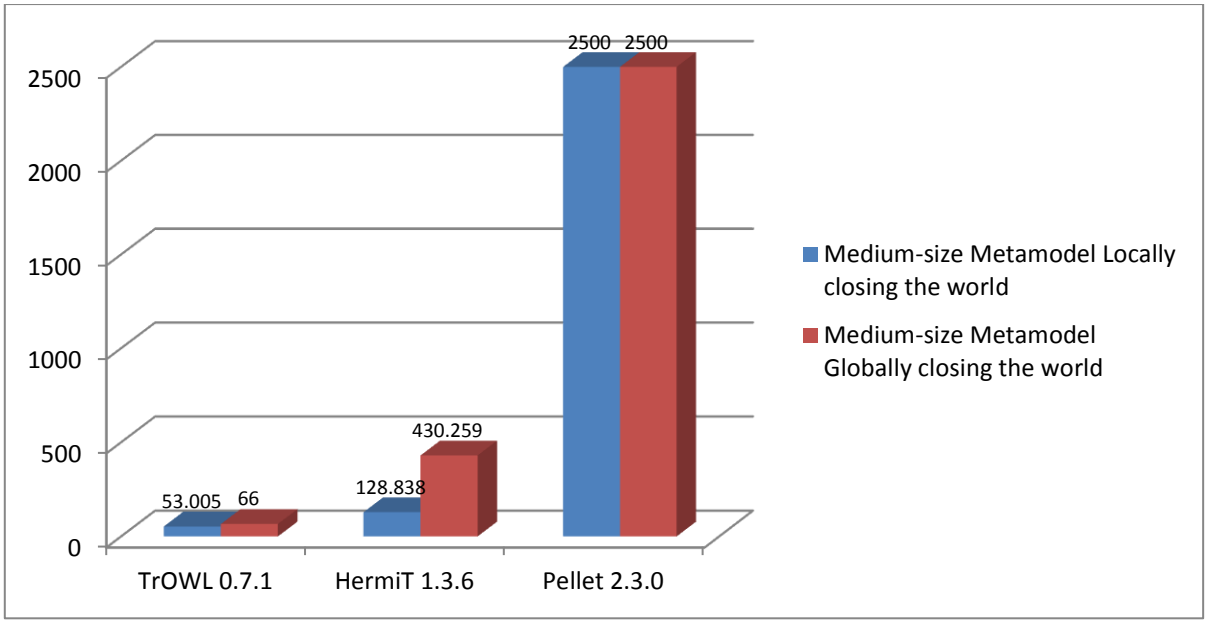


Figure 6-8: Correlation of globally/locally closing the world and reasoners performance medium-size ontology

The bar chart above refers to the medium-size ontology and once again represents the effect of locally closing the world, in comparison to globally closing the world, upon the reasoning performance of all three semantic reasoners of the ontology. Each of the three sets corresponds to a semantic reasoner. The first set corresponds to the TrOWL 0.7.1 reasoner, the second one to the HermiT 1.3.6 and the third set to the Pellet 2.3.0 reasoner. Every set consists of two differently coloured bars each of which corresponds to the applied approach. The blue one represents the ontology where the open world has been locally closed. The red one represents the second alternative approach, where the open world has been globally closed.

In the case of the medium-size ontology, TrOWL reasoner seems once again to handle both approaches quite effectively, as there is no difference upon the reasoning time. HermiT's reasoning performance though seems to be approx. three times slower when the second approach is applied, in comparison to the first one, where the open world is locally closed. Pellet reasoner finally, once again didn't manage to complete reasoning of the ontology regardless of the approach.

Regarding the large-size ontology, none of the semantic reasoners managed to complete reasoning within the time limit that was set, and thus no conclusions could be withdrawn.

6.6 Lessons Learned

Main objective throughout this underlying work has been from the very beginning, apart from establishing our approach and ensuring its correctness to succeed the most effective and accurate validation of ADOxx metamodels transformed into OWL 2 ontologies by succeeding the best possible reasoning performance. This, among many other factors, has also been one of the major factors regarding the definition of our approach, that in many cases influenced at a great extent the chosen approach to be applied, in case that more than one have been developed.

There have been not a few cases that reasoners performance and capability of dealing effectively with the given ontologies was the only factor that guided us, and influenced our final decision regarding the choice of alternative approaches. Having developed a number of different approaches and implementations, and having performed various performance comparisons regarding semantic reasoners, we came to the following conclusions:

- Ontology size and more specifically the size of ontology ABox does affect the reasoning performance. More specifically, the bigger the ABox of the ontology is, the more it takes for the reasoning to be completed. According to our comparison results, reasoning time seemed to increase exponentially for an average increase of three thousand new instances of the ABox.
- When it comes to choose the OWL 2 expressivity, and more specifically between SROIQ and SROIQ (D), the first alternative seems to succeed a better reasoning performance in comparison to the second one. The use of data properties seems to double the time that reasoning needs to be completed, or even multiply it in case of some semantic reasoners (Pellet 2.3.0).
- The OWA and its principle that non-existence is not interpreted as wrong statement, seems to fit perfectly the unlimited nature of knowledge. In the case though that closing the open world is a necessity, then closing the open world locally seems to be more effective in comparison to a global approach. More particularly, in case of closing the open world globally, reasoning performance increases multiple times from that succeeded when closing the world locally.
- The use of DL safe rules for expressing concepts that cannot be expressed with the use of OWL 2 axioms (e.g. concept cyclicity) extends at a great degree the expressive power of OWL 2. Their high complexity though affects reasoning performance, by slowing it down multiple times.
- In the case that OWL 2 *EquivalentClass* axioms are used, their definition should be as simple and concrete as possible. Vague, complex or not well constructed OWL 2 axioms, make reasoning process quite time consuming process.

7 Conclusions & Future Work

The potential synergy between semantic technologies and metamodelling area, and the lack of such approaches triggered the realisation of this underlying work. Throughout this work, there has been a thorough attempt to investigate the use of semantic technologies as a means to ensure the correctness of metamodels. In this investigation, the aim was to assess the applicability of semantic technologies as a means for the definition of ADOxx meta-metamodel main concepts and of its semantic constraints, conformance to which ensures the correctness of metamodels.

For this purpose, this study had to be divided into three main tasks:

- Definition of a mapping approach for converting a given meta-metamodel, and its metamodels into an OWL 2 ontology
- Establishment of an approach for defining a given set of meta-metamodel semantic constraints in OWL 2
- Development of a validation mechanism for implementing semantic checks and identifying/detecting the violation of semantic constraints

For the realisation of these tasks, a metamodelling tool with a concrete meta-metamodel was needed. The meta-metamodel should be tightly coupled with a well-defined set of semantic constraints to which metamodels should conform to, and should provide a validation mechanism for detecting their violation. ADOxx metamodelling platform, a metamodelling tool developed by BOC group gathered all these features, and thus was used for the realisation of this work andall its sub-tasks.

The factors that had to be taken into consideration throughout the whole process can be summarized as follows:

- Correct translation of ADOxx meta-metamodel into OWL 2
- Correct and precise definition of ADOxx semantic constraints into OWL 2
- Effective and correct validation of ADOxx semantic constraints

Regarding the definition of a mapping approach for converting the ADOxx meta-metamodel, and a given set of ADOxx metamodels into an OWL 2 ontology, our study has proved the applicability of this approach. More precisely, it has been proved that ontology's structure and its main building blocks, together with the ontology language OWL 2, provide all the necessary means and the needed expressive power for realising this approach. The realisation of this task has shown that the definition of the mapping approach was not a one-to-one mapping process, as most of the times there has been more than one alternative approaches for the mapping approach.

The whole mapping process has been iterative due to the arising factors that should be taken into consideration or be overcome. The findings of this process suggest that OWA, although it serves the concept of knowledge and its limitless nature, it can create a number of

complications, when a DBMS system is used, and Negation as Failure is needed (section 2.2.3.2). NaF, being closely related to the CWA, made us experiment and apply an approach for locally closing the open world.

The misconception of the concept "same as" and its implications was another finding that came out during the definition of the mapping approach. Defining OWL 2 individuals to be the same, does not always imply their equivalence. In our case, assuming that these instances are identical proved to be wrong, as they turned out to be identical but referentially opaque, meaning that they do identify the same thing, but all the properties ascribed to one individual are not necessarily the same (e.g. the default value).

Regarding the integration of ADOxx semantic constraints into the ontology representing the ADOxx meta-metamodel, two main factors needed to be specified and studied: OWL 2 expressive power, and the adequacy of reasoning services for contributing into the implementation of a semantic constraint validation mechanism.

Classification of ADOxx semantic constraints was the outcome of our attempt to deal with their big number and to identify reoccurring constraints. Main objectives were to simplify the process of defining semantic constraints, to reduce the risk of defining contradictory ones and to lessen their complexity.

The first classification of ADOxx semantic constraints was based on whether they are enforced by the metamodel editor, or not. Semantic constraints enforced by the editor refer to ADOxx core element properties, and were further classified into those that refer to mutually exclusive properties, and those related with the relationship "sub-property". Semantic constraints not enforced by the editor refer to ADOxx library checks and are further classified into quantitative and qualitative ones.

Another significant finding to emerge from this task was an analysis of the services provided by semantic reasoners, and how they could be used for detecting a potential violation of semantic constraint. Checking ontology's consistency was the first reasoning service to be studied. An extensive analysis showed that ontology's consistency checking lacksthe ability to provide a description of the reasons that lead to an inconsistency, and thus to the violation of a constraint. Classification was the second reasoning service to be analysed. Our study showed that this reasoning service allows the use of semantic annotations for enriching the description of the reasons that lead to the violation of a constraint, but it can only be exploited with the use of necessary and sufficient OWL 2 axioms. The final approach exploits both reasoning services, depending on the nature of the constraints.

The results of the research carried out within this second task imply that OWL 2 is capable of defining the majority of concepts that appear within ADOxx semantic constraints. Major exception is considered to be the case when constraints refer to a conceptual cyclicity. For this, the use of SWRL is required.

Regarding the third task, main focus lied on the development of a validation mechanism for reasoning the constructed ontology and identifying/detecting the violation of semantic constraints. One of the major findings was the importance and usefulness of semantic annotations. Within our study we showed how semantic annotations can be exploited so as to

achieve the lowest possible loss of information during the conversion of the ADOxx meta-metamodel to an ontology. Semantic annotations can be used for enriching the ontology with humanly meaningful information regarding the main ADOxx concepts and relations, and the ADOxx semantic constraints. Last but not least, we showed how they can be used for reasoning purposes with main objective to retrieve the set of individuals that violate specific ADOxx semantic constraints.

Last task of this underlying work was an evaluation of the currently used semantic reasoners regarding the factors that influence their performance. Our study focused on the correlation between reasoning performance and the following four factors:

- Ontology size
- Use of OWL 2 data properties
- Use of rules
- Closing the open world

Regarding ontology size, and more specifically the size of ontology ABox, our study showed that the time needed for completing the reasoning process increases exponentially as the size of the ABox increases. The use of OWL 2 data properties, although it can be a necessity, it was proved to slow down reasoning performance. It is not argumentable that semantic rules, and their built-in axioms, contribute in enriching expressive power of OWL 2. Their complexity though makes their use and appliance prohibitive, as it has a great effect on reasoning performance. When, finally, closing the open world is considered to be a necessity, our study showed, that locally closing the open world can extenuate the reasoning performance, in contrast to globally closing the world, which was proved to slow down the performance at a great point.

The underlying work confirms previous findings and contributes additional evidence that argue for the use of the ontology language OWL 2 for ensuring correctness of metamodels. Empirical findings in this study provide a new understanding regarding how a concrete meta-metamodel and a set of metamodel instances can be mapped to an ontology, how OWL 2 can be used for defining semantic constraints, how reasoning services and OWL 2 can be combined for developing a validation mechanism for detecting constraint violations, and finally which factors affect reasoning performance, and how it can be improved.

Currently, the time needed for completing the validation of a medium-size metamodel within ADOxx metamodelling platform is approximately one second. The reasoning process of a medium-size ontology with the current semantic reasoners is completed approximately in one minute. This performance can be considered to be prohibitive regarding the use of semantic technologies for ensuring correctness of metamodels within real-world industrial projects. However, taking into consideration the intensive research into this field, and the impressive progress that has been achieved the last decade concerning the semantic technologies and their appliance upon a number of research fields, this leaves plenty of room for believing that soon enough semantic technologies will be widely used in industry for ensuring correctness of metamodels.

Based on the findings of this work, further research could take place into how could DL-SPARQL be used as a query language, not only for the explicitly stated, but also for the implicit knowledge, how semantic reasoners could cope with complexity of rules more effectively, and how incremental reasoning could be extended from the TBox to the ABox, as well.

Table of Abbreviations

Bibliography

(BSSC), ESA Board for Software Standardisation and Control. *Guide to the software requirements definition phase.* The Netherlands: ESA Publications Division, 1995.

2011 Semantic Technology Conference. June 2011. http://semtech2011.semanticweb.com (accessed July 07, 2012).

Alloy Constraint Language. n.d. http://alloy.mit.edu/alloy/ (accessed June 01, 2012).

An Oracle White Paper, Getting Started With UML Class Modeling. May 2007. http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html (accessed March 03, 2012).

Antoniou, Grigoris, and Frank van Hamerlen. "Web Ontology Language: OWL." In *Handbook on Ontologies*, 91-110. Springer-Verlag Berlin Heidelberg, 2009.

Antoniou, Grigoris, and Frank Van Hamerlen. "Web Ontology Language: OWL." In *Handbook on Ontologies in Information Systems*, 67-92. Springer, 2003.

Aßmann, Uwe, Jürgen Ebert, Tobias Walter, and Christian Wende. "Ontology and Bridging Technologies." In *Ontology-Driven Software Development*, by Jeff Z. Pan, Steffen Staab, Uwe Aßmann, Jürgen Ebert, & Yuting Zhao, 179-192. Springer-Verlag Berlin Heidelberg, 2013.

Baader, Franz, and Werner Nutt. "Basic description logics." In *The description logic handbook*, 43-95. New York, USA: Cambridge University Press New York, 2003.

Baader, Franz, Diego Calvanese, Deborah L. McGuiness, Daniele Nardi, and Peter F. Patel-Schneider. *The description logic handbook: theory, implementation, and applications.* New York, USA: Cambridge University Press, 2003.

Baader, Franz, Ian Horrocks, and Ulrike Sattler. "Description Logics." In *Handbook on Ontologies*, 21-44. Springer-Verlag Berlin Heidelberg, 2009.

Baar, Thomas. "Correctly defined concrete syntax for visual modeling languages." *MoDELS'06 Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems.* 2006.

Bézivin, Jean. "On the unification power of models." In *Software and System Modeling*, 171-188. 2005.

BOC Group. *ADOxx metamodelling platform.* n.d. http://www.adoxx.org/live/ (accessed April 05, 2013).

Booch, Grady, Ivar Jacobson, and James Rumbaugh. *UML Semantics (Version 1.1).* Rational Corporation, Santa Clara, 1997.

Boronat, Artur, and José Meseguer. " Algebraic Semantics of EMOF/OCL Metamodels." 2007.

Cadavid, Juan, Benoit Baudry, and Benoit Combemale. "Empirical evaluation of the conjunct use of MOF and OCL." *Proceedings of EESSMOD workshop at MODELS'11.* Wellington, New Zealand, 2011.

Cockburn, Alistair. "Using Both Incremental and Iterative Development." *CROSSTALK The Journal of Defense Software Engineering*, May 2008: 27-30.

ConceptBase cc. *ConceptBase.cc - A Database System for Metamodeling and Method Engineering.* n.d. http://conceptbase.sourceforge.net/ (accessed June 20, 2013).

Fensel, Dieter, Frank Van Hamerlen, Ian Horrocks, Deborah L. McGuinness, and Peter F. Patel - Schneider. "OIL: An ontology infrastructure for the semantic web." *IEEE Inteligent systems* 16, no. 2 (2001): 38-45.

Garcia, Daniel, Catalina M. Llado, Connie U. Smith, and Ramon Puigjaner. "Performance Model Interchange Format: Semantic Validation." *ICSEA '06 Proceedings of the International Conference on Software Engineering Advances.* Washington, DC, USA, 2006.

Garcia, Miguel. "Rules for Type-checking of Parametric Polymorphism in EMF Generics." *GI-Edition Lecture Notes in Informatics* 106 (2007): 261-270.

Gerber, Aurona, Alta van der Merwe, and Andries Barnard. "A functional semantic web architecture." *Proceedings of the 5th European semantic web conference on The semantic web: research and applications.* Tenerife, Canary Islands, Spain: Springer-Verlag, 2008. 273--287.

Gogolla, Martin, and Mark Richters. "Equivalence rules for UML class diagrams." *Proceedings of UML'98 International Workshop.* Mulhouse, France, 1998.

Gómez-Pérez, Asunción, and Oscar Corcho. "Ontology Languages for the Semantic Web." *IEEE INTELLIGENT SYSTEMS* January/February 2002 (2002): 54-60.

Gruber, Thomas R. "A translation approach to portable ontology specifications." *KNOWLEDGE ACQUISITION*, 1993: 199--220.

Halpin, Harry, Patrick J. Hayes, James P. McCusker, Deborah L. McGuinness, and Henry S. Thompson. "When owl:sameAs Isn't the Same: An analysis of identity in Linked Data." *International Semantic Web Conference.* 2010.

Harel, David, and Bernhard Rumpe. *Modeling Languages: Syntax, Semantics and all that Stuff (or, What's the Semantics of "Semantics"?).* Technische Universität Braunschweig, 2004.

Heflin, Jeff, James Hendler, and Sean Luke. *SHOE: A Knowledge Representation Language for Internet Applications.* University of Maryland, Department of Computer Science, 1999.

Hitzler, Pascal, and Bijan Parsia. "Ontologies and Rules." In *Handbook of Ontologies*, 111-132. Springer-Verlag Berlin Heidelberg , 2009.

Horridge, Matthew, and Sean Bechhofer. "The OWL API: A Java API for Working with." *Proceedings of OWL: Experiences and Directions.* 2009.

Horridge, Matthew, Johannes Bauer, Bijan Parsia, and Ulrike Sattler. "Understanding Entailments in OWL." *Proceedings of the Fifth OWLED Workshop on OWL: Experiences and Directions, collocated with the 7th International Semantic Web Conference (ISWC-2008).* 2009.

Horrocks, Ian, Peter F. Patel - Schneider, and Frank Van Harmelen. "From SHIQ and RDF to OWL: The Making of a Web Ontology Language." *Journal of Web Semantics* 1, no. 1 (2003): 7-26.

Institute for Software Integrated Systems. *GME Metamodellig Environment.* n.d. http://w3.isis.vanderbilt.edu/Projects/gme/meta.html (accessed June 20, 2013).

Karagiannis, Dimitris, and Harald Kühn. "Metamodelling Platforms." *Proceedings of the 3rd international conference ec-web 2002 - dexa 2002.* aix-en-provence, france: Springer-Verlag, 2002. 182.

Karagiannis, Dimitris, and Peter Höfferer. "Metamodels in action: An overview." *ICSOFT 2006, First International Conference on Software and Data Technologies.* Setúbal, Portugal: INSTICC Press, 2006.

Karagiannis, Dimitris, Hans-Georg Fill, Srdjan Zivkovic, and Wilfrid Utz. "From Model Editors to Modelling Tools: Operationalizing Modelling Methods with ADOxx." *ACM/IEEE 15th International Conference on Model-Driven Engineering Languages and Systems (MODELS'2012).* Innsbruck, 2012.

Kelsen, Pierre, and Qin Ma. "A Lightweight Approach for Defining the Formal Semantics of a Modeling Language." *MoDELS '08 Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems* . 2008.

Kent, Stuart. "IFM '02 Proceedings of the Third International Conference on Integrated Formal Methods ." London, UK: Springer-Verlag, 2002. 286-298 .

Kleppe, Anneke. “A Language Description is More than a Metamodel.” *Fourth International Workshop on Software Language Engineering.* Nashville, USA, 2007.

Kollia, Ilianna, Birte Glimm, and Ian Horrocks. “Query Answering over SROIQ Knowledge Bases with SPARQL.” *2011 International Workshop on Description Logics (DL2011).* 2011.

Kühn , Harald, and Marion Murzek. “Interoperability Issues in Metamodelling Platform.” In *Interoperability of Enterprise Software and Applications*, 215-226. Springer Verlag, 2006.

Kühn, Harald, Franz Bayer, Stefan Junginger, and Dimitris Karagiannis. “Enterprise Model Integration.” *Proceedings of the 4th International Conference EC-Web 2003.* Prague, Czech Republic: Springer Verlag, 2003. 379-392.

Ledeczi, Akos, et al. “The Generic Modeling Environment.” *Workshop on Intelligent Signal Processing at WISP’2001.* Budapest, Hungary, 2001.

Linthicum, David S. *Enterprise Application Integration.* Addison-Wesley Professional, 2000.

Loecher, Sten, and Stefan Ocke. “A Metamodel-Based OCL-Compiler for UML and MOF.” *Electronic Notes in Theoretical Computer Science (ENTCS)* 102, no. November (2004): 43-61.

Metacase. *Metacase.* n.d. http://www.metacase.com/ (accessed June 2013, 21).

Obeo Designer. *Obeo Designer.* n.d. http://www.obeodesigner.com/ (accessed June 2013, 20).

Object Management Group (OMG). *Meta Object Facility (MOF) Specification 1.4.* 2001. http://www.omg.org/spec/MOD/1.4/PDF (accessed February 06, 2012).

—. *Object Constraint Language (OCL), version 2.2.* February 2010. http://www.omg.org/spec/OCL/2.2/PDF/ (accessed June 06, 2012).

Object Management Group. *MDA Specifications.* n.d. http://www.omg.org/mda/specs.htm (accessed February 08, 2012).

—. *Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT).* January 2011. http://www.omg.org/spec/QVT/ (accessed June 01, 2011).

—. *Unified Modeling Language Specification, OMG Specifications.* September 2003. http://www.omg.org/docs/formal/03-03-01.pdf (accessed September 16, 2012).

Ontology. n.d. http://www.emc-eu.de/index-Dateien/3_ONTOLOGY_UK.html (accessed October 26, 2012).

OWL 2 Web Ontology Language Manchester Syntax, W3C Working Group Note. 27 October 2009. http://www.w3.org/TR/owl2-syntax/#Class_Expressions (accessed June 28, 2012).

OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax. n.d. http://www.w3.org/TR/owl2-syntax/#Axioms (accessed November 02, 2012).

Paige, Richard F., Phillip J. Brooke, and Jonathan S. Ostoff. "Metamodel-based model conformance and multiview consistency checking." *ACM Transactions on Software Engineering and Methodology (TOSEM)* 16, no. 3 (2007): Article No. 11.

Petrascu, Vladiela, and Dan Ioan Chiorean. "Proposal of a Set of OCL WFRs for the ECore." *Studia Universitatis Babes-Bolyai : Series Informatica* 54, no. 2 (2009): 89.

Petrascu, Vladiela, and Dan Ioan Chiorean. "Towards Improving the Static Semantics of XCore." *Studia Universitatis Babes-Bolyai : Series Informatica* 55, no. 3 (2010): 61.

Pohjonen, Risto, and Juha-Pekka Tolvanen. "SPLC'05 Proceedings of the 9th international conference on Software Product Lines." *Springer-Verlag Berlin, Heidelberg.* Jyväskylä, Finland, 2005.

Shearer, Rob, Boris Motik, and Ian Horrocks. "HermiT: A Highly-Efficient OWL Reasoner." *Proc. of the 5th Int. Workshop on OWL: Experiences and Directions (OWLED 2008 EU).* Karlsruhe, Germany, 2008.

Siddiqui, Awes, and Rucha Deshmukh. "SEMANTIC WEB TECHNOLOGY." *Bioinfo Publications*, 2012: 20-23.

Sirin, Evren, Bijan Parsia, Bernando Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. "Pellet: A Practical OWL-DL Reasoner." *Web Semantics: Science, Services and Agents on the World Wide Web* 5, no. 2 (2007): 51-53.

Staab, Steffen, and Rudi Studer. "What Is an Ontology?" In *Handbook on Ontologies*, 1-2. Springer-Verlag Berlin Heidelberg, 2009.

Steinberg, David, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2009.

Struder, Rudi, Richard Benjamins, and Dieter Fensel. "Knowledge Engineering: Principles and methods." *Data & Knowledge Engineering* 25 (1998): 161-197.

Thomas, Edward, Jeff Z. Pan, and Yuan Ren. "TrOWL: Tractable OWL 2 Reasoning Infrastructure." In *The Semantic Web: Research and Applications*, 431-435. Springer-Verlag Berlin Heidelberg, 2010.

Tu, Kewei, Miao Xiong, Lei Zhang, Haiping Zhu, Jie Zhang, and Yong Yu. “Towards Imaging Large-Scale Ontologies for Quick Understanding and Analysis.” *Proceedings of the Fourth International Semantic Web Conference (ISWC2005).* 2005. 702--715.

W3C. *SPARQL Query Language for RDF*. 15 January 2008. http://www.w3.org/TR/rdf-sparql-query/ (accessed January 16, 2013).

Walter, Tobias, Fernando Silva Parreiras, and Steffen Staab. “OntoDSL: An Ontology-Based Framework for Domain-Specific Languages.” *MODELS '09 Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems* . 2009.

Zedlitz, Jesper, Jan Jörke, and Norbert Luttenberger. “From UML to OWL 2.” In *Knowledge Technology*, 154-163. Springer Berlin Heidelberg, 2012.

Zivkovic, Srdjan, Harald Kühn, and Marion Murzek. “An Architecture of Ontology-aware Metamodelling Platforms for Advanced Enterprise Repositories.” Milano, Italy, 2009.

Zivkovic, Srdjan, Marion Murzek, and Harald Kühn. “Bringing Ontology Awareness into Model Driven Engineering Platforms.” *1st International Workshop on Transforming and Weaving Ontologies in Model Driven Engineering TWOMDE*. Toulouse, France, 2008. 47-54.

Curriculum Vitae

Christos Lekaditis

Studies

09/2003 - 02/2009: National and Kapodistrian University of Athens –

School of Sciences –

Informatics and Telecommunications (BSc)

Main topics:

- Protection of security systems
- Innovation and business dexterity
- Human-machine interaction
- Information system analysis
- Algorithm operational research

Thesis Title:

- Experimental selection of optimal device emulating a mouse

10/2009 – currently Universität Wien –

Wirtschaftsinformatik (MSc)

Thesis Title:

- Validation of ADOxx metamodels based on Semantic Technologies

Abroad for study purposes (Erasmus, Joint Studies, Fulbright etc.)

07/2010:	Participation at "IPICS 2010" Summer School (Intensive Programme on Information and Communication Security) at University of Aegean, Samos, Greece (http://www.ipics-school.eu/)
03/2007 – 06/2007:	Erasmus Programme - University of Vienna - Faculty of Informatics
03/2008:	Participation at "IPICS 2008" Winter School (Intensive Programme on Information and Communication Security) at University of Lapland, Rovaniemi, Finland (http://www.ipics-school.eu/)

Further Qualifications

Greek

English

German