Master's thesis

An Ontology-Based Metalanguage with Explicit Instantiation

By
Alfons Laarman

University of Twente

Faculty of Electrical Engineering, Mathematics and Computer Science
Department of Computer Science
Software Engineering Group

29 March 2009

Committee members:
dr. Ivan Kurtev
dr. ir. Klaas van den Berg
Arda Goknil, MSc.

Abbreviations

General Terminology

Abbreviation	Full Description
AST	Abstract Syntax Tree
BPEL	Business Process Execution Language
BNF	Backus Naur Form
BWW	Bunge-Wand-Webber (ontology)
CIM	Computational Independent Model
CWM	Common Warehouse MetaModel
DSL	Domain-Specific Language
EBNF	Extended Backus Naur Form
ECore	EMF Core
EMF	Eclipse Modeling Framework
FCO	Four-category ontology
KM3	Kernel MetaMetaModel
MD	Moment Definition
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MOF	Meta Object Facility
OCL	Object Constraint Language
OGML	Ontology Grounded MetaLanguage
OGMLX	OGML eXtensional
OWL	Web Ontology Language
PIM	Platform Independent Model
PSM	Platform Specific Model
RDF	Resource Description Framework
SE	Software Engineering
UML	Unified Modeling Language
XML	eXtensible Markup Language

OGML Specific Terminology

Abbreviation	Full Description
AF	AttributeFunction
CI	CharacterizationInstantiation
CR	CharacterizationRelation
DTD	DataTypeDefinition
ID	IndividualDefinition
IE	IdentifiableElement
IOP	InstanceOfProperty
IOD	InstanceOfDefinition
IR	InherenceRelation
LD	LanguageDefinition
MD	MomentDefinition
ME	ModelElement
MM	MetaModel
MU	MomentUniversal

OD	ObjectDefinition
PD	PropertyDefinition
PE	PropertiesElement
SD	SubstantialDefinition
SU	SubstantialUniversal
UD	UniversalDefinition
XO	XObject

Abstract

Model Driven Engineering (MDE) is a promising paradigm for software development. It raises the level of abstraction in software development by treating models as primary artifacts. The practical application of this paradigm is seriously endangered by the current weak modeling foundation of the approach. The objective of this work is to provide a sound foundation to modeling in MDE based on Formal Ontologies.

MDE advocates the definition of the abstract syntax of modeling languages, both domain-specific and general-purpose, by means of metamodels. Metamodels are expressed in a specialized metalanguage. The definition of a metamodel is a recurring task in MDE and requires sound and formal support. The lack of such support causes deficiencies such as conceptual anomalies in the modeling languages, limited applicability of crucial MDE operations like model transformation and querying, and limited reuse of model libraries.

From philosophical point of view, metamodels can be seen as ontological commitments. Metalanguages have to provide constructs for building ontological theories as a base for modeling languages. In this thesis, we present a new metalanguage derived from the study of Formal Ontology. This metalanguage, called OGML, raises the level of abstraction of metamodels from pure abstract syntax to semantics descriptions based on ontologies. Thus, the language developers can make conscious choices for their modeling concepts and can explicitly define important relations such as instantiation. With this metalanguage, we aim at a precise conceptual and formal foundation for metamodeling.

OGML is capable of expressing metamodels and models together with the instantiation semantics of the language. In current tools, these relations are not uniformly treated for different languages. Our metalanguage provides the means to do this. Thereby it makes the concepts that play a role in metamodeling more explicit and lifts some limitations of traditional metalanguages. We have shown the capabilities of the metalanguage by expressing several modeling languages in it. The result is that we are able to handle all languages and their models uniformly. An OCL interpreter has been created to demonstrate that. The OCL interpreter allows the navigation on models both according to the semantics of their different modeling languages and the metalanguage. A model conformance checker is provided to verify correctness of a model according to another model in the same language, but also to verify models and languages against the metalanguage.

The architecture of our metalanguage is centered around the concept of modeling languages. Each language defines its own notion of instantiation. The metalanguage is also a modeling language, so it also has a notion of instantiation. In our architecture, we can treat all instantiation mechanisms uniformly, which is not the case in traditional metalanguages like MOF, EMF and KM3.

Samenvatting

Model Driven Engineering (MDE) is een veelbelovend paradigma voor het ontwikkelen van software. Het verhoogt het abstractieniveau in de softwareontwikkeling door modellen als primaire artefacten te gebruiken. De praktische toepasbaarheid van dit paradigma wordt echter bedreigd door de zwakke conceptuele basis. Het doel van dit werk is om een correcte ondergrond voor modelleren in MDE te creëren met behulp van formele ontologieën.

MDE pleit ervoor om de definitie van de abstracte syntaxis van zowel domeinspecifieke als ook generieke, modelleertalen uit te drukken door middel van metamodellen. Metamodellen worden op hun buurt uitgedrukt in metatalen. De definitie van een metamodel is een veelvoorkomende taak binnen MDE. Deze taak vereist correcte en formele ondersteuning. Het gebrek daaraan veroorzaakt tekortkoming, zoals conceptuele afwijkingen in modelleertalen, gelimiteerde toepasbaarheid van cruciale MDE-operaties (modeltransformaties en queries) en gelimiteerd hergebruik van modelbibliotheken.

Vanuit filosofisch oogpunt kunnen metamodellen worden gezien als "ontologic commitment", een ontologische aanname. Metatalen moeten de juiste bouwstenen aanbieden voor het bouwen van ontologische constructies als een basis voor modelleertalen. In deze thesis presenteren wij een metataal, die afgeleid is van het onderzoeksgebied Formele Ontologie. Deze metataal verhoogt het abstractieniveau van metamodellen van pure definities voor de abstracte syntaxis tot semantische beschrijvingen gebaseerd op ontologieën. Daardoor kunnen taalontwikkelaars bewuste keuzes maken voor hun modelleerconcepten en kunnen belangrijke relaties, zoals de instantiatie-relatie, expliciet beschreven worden. Met deze metataal hebben we als doel om een conceptuele en formele grondslag te leveren voor metamodellering.

The metataal die hier wordt gepresenteerd is geschikt om metamodellen en modellen uit te drukken samen met de semantiek voor instantiatie van de taal. In huidige tools worden deze relaties niet gelijkmatig behandeld. Onze metataal biedt de mogelijkheden om dit wel te realiseren. Daardoor worden de concepten, die een rol spelen in metamodellering, explicieter en worden enkele limitaties van traditionele metatalen opgeheven. We hebben de mogelijkheden van deze taal aangetoond door er meerdere modelleertalen in uit te drukken. Een OCL-"interpreter" is gemaakt om te demonstreren dat dit resulteert in uniforme verwerking van de instantiatie-relaties over meerdere talen. Deze "interpreter" staat het toe om te navigeren over modellen in overeenstemming met de semantiek van hun verschillende modelleertalen en tevens van de metataal. Een "checker" is gerealiseerd om te kunnen controleren of modellen conform zijn aan andere modellen in overeenstemming met de semantiek van de modelleertaal, maar ook of de talen conform zijn aan de metataal.

De architectuur van onze metataal is gebouwd om het concept modelleertaal. Elke taal definieert zijn eigen zicht op het instantiatie-mechanisme. De metataal is ook een modelleertaal en heeft dus ook een zicht op instantiatie. In onze architectuur kunnen we al deze instantiatie-mechanismen uniform behandelen. Dat is niet het geval bij traditionele metatalen zoals MOF, EMF en KM3.

Table of Contents

List of Figures

List of Tables

List of Code Listings

Acknowledgements

I would like to thank the people that helped me realize this work.

In the first place, I thank Ivan Kurtev for taking me under his supervision. He allowed me the freedom to choose an interesting subject and gave inspiring explanations to the topics. I learned many things during the months that he guided me. When I was working out the small details making my own choices, Ivan always had a high-level view of the process and often predicted the outcome. Any disagreement soon proved me wrong in practice. In this way, I learned to think on the meta-level as well.

Ivan showed great flexibility in working with his students allowing me to still have fun even in the busiest periods of these last months. His provocative remarks in "het afstudeerhok" often induced great laughter amongst us students. I sincerely hope that Ivan is able to pursue his career objectives in the future and at the same time can keep the joy in his work and life.

I would also like to thank Klaas van den Berg and Arda Goknil for agreeing to be member of the graduation committee. Klaas helped me a lot by reviewing this work. The result certainly became more readable and consistent due to his comments.

I thank my fellow students with whom I spent the last year of my study.

Especially I thank Mark Timmer, Lucas Meertens and Jan-Willem Veldhuis for reviewing this work. Mark thank you for your detailed comments in the final phase of my work. Lucas also agreed to read and comment on my work several times. Jan-Willem was able to help me by asking the right questions and he commented on this thesis. I wish you all good luck with the career you are starting right now.

I would also like to thank Jaco van de Pol, Michael Webber and Mariëlle Stoelinga for providing me the opportunity to work as a PhD student in Enschede. Mariëlle made me aware of the position. I feel grateful that these people saw me fit for the job.

Thanks Laura, for your love, support and confidence.

Als laatste, maar natuurlijk voor mij als belangrijkste, dank ik mijn ouders voor hun ondersteuning en advies. Zonder hun was ik hier niet gekomen. Robbin en Fabian bedankt voor jullie ondersteuning. Succes, dan drinken we binnenkort op jullie afstuderen.

Chapter 1 – Introduction

Software systems grow larger every day. To keep their implementations manageable, portable and understandable, it is often beneficial to raise the level of abstraction above the implementation technology level. Abstraction provides a good basis to cope with evolution, which is ever more crucial in today's fast-growing software systems. Not only do these systems grow more dependent on other (software) systems, they are also subject to change of the software technologies used for their implementation.

The Object Management Group (OMG), a consortium of software industry participants and academia, provides a software development approach based on models: Model Driven Architecture (MDA). MDA promotes the use of models in software development. By treating models as primary artifacts, it raises the level of abstraction in software development and emphasizes on the activity of modeling.

Models are abstractions of reality made for communication, documentation and analysis. Model Driven Engineering (MDE) [55] describes how models are used in the context of a software development process. Compared to MDA, MDE takes a more general approach to modeling by including different technologies like databases and XML. A recurrent activity in MDE is the definition of models on the abstract level and the derivation of a more detailed, more concrete, model. This can be done in an automated fashion with, for example, a model transformation language. The abstract layer can be the conceptualization of objects that exists in a system, and the more concrete layer can be an implementation in a programming language representing these conceptualizations.

In this thesis, we are interested in models expressed in a modeling language. We observe usage of both general-purpose and domain-specific modeling languages (DSLs) in the current practice. A well-known general-purpose modeling language is UML. However, domain-specific modeling languages are gaining increasing popularity. For example, [54] reports on shortened development time and reduced cost in several industrial projects that employ DSLs. Software industry also started offering tool sets for DSL development like Microsoft DSL tools [38], Intentional programming [49], and Eclipse Modeling Framework (EMF) [28]. As a result, modeling language development has also become a recurring activity in MDE.

1.1 Preliminaries

The traditional approach to define a language is to first specify its grammar. MDE takes a different approach by using a *metamodel* to define the abstract syntax of a language. This is applied for both general-purpose and domain-specific modeling languages. Furthermore, in MDE, programs and user data are models [17], that way allowing programming and data description languages to be defined by metamodels as well.

A metamodel defines the structure of the admissible models in a language. In an activity called *metamodeling,* metamodels are treated as models themselves and are expressed in a modeling language. Such a language is known as a *metalanguage*. The metamodel of a

metalanguage is termed *metametamodel*. Most current MDE approaches provide object-oriented *metametamodels* such as MOF [71], KM3 [50], and EMF. The reason for this is pragmatically motivated: object-orientation is the dominant software development paradigm today supported by mature tools.

The stack of metametamodel, metamodels and models is called a *modeling architecture*. OMG's MOF architecture defines this stack as a strictly layered design. This means the models of each layer only conform to the layer directly above it. The layers are called M3 for metametamodels, M2 for metamodels and M1 for models. A model from layer M1 is an *instanceOf* a model of layer M2 and M2 of M3. This relation between layers is also called the *instantiation relation*.

UML is a modeling language used at layer M2 in the modeling stack. It is intended to be a general-purpose language but focuses especially on software design. In order to introduce constraints on its instances, both UML and MOF definitions can contain OCL code. OCL is an expression language able to navigate models and define constraints over them.

M0 is added to carry user data models directly representing things in the real world. In later versions of MOF, the user data models were incorporated in the M1 layer. The M0 layer is assumed to be the real world [7].

1.2 Problem Statement

In the literature about MDE, we found several problems attributed to the modeling architecture. We believe their causes are deeply rooted in the definition of this architecture. To investigate the problems we define it as the difference between the needs and the state of the art. The application of MDE promises several benefits for software development:

- Models can provide a *consistent* and *unambiguous* way to represent knowledge about a system or a domain. This can improve the efficiency and precision of crucial activities like system design, communication and documentation,
- To capture *domain-specific* knowledge, a modeler can specify his own DSL in the form of a metamodel. This allows the models to be interpreted in a specialized manner, which offers benefits for the application of MDA [54]. *Cross language interoperability* should ensure the uniform handling of models conforming to different metamodels,
- Models can be *interpreted* by a machine, paving the way for the *automation* of development activities. For example, the design of a system can be combined with knowledge about the implementation technology to generate automatically a complete or partial implementation.

The ability to reap these benefits of MDE depends on the correctness and consistency of the models specified by the modelers themselves. However, it also depends on the ability of a modeling architecture to treat the different metamodels and models in a generic fashion. This in turn depends on the metalanguage, which therefore is crucial for the whole modeling architecture. In this thesis, we focus on exactly those technologies and intend to improve the currently available solutions.

Problems in MDE

In the light of the described promises of MDE, we perceive several problems with traditional modeling architectures:

Expressiveness problems – Recent studies on UML [42][43] showed several inadequacies of this language regarding its modeling foundation such as *construct incompleteness*, *construct overloading*, and *construct redundancy*. This might result in *inconsistent* and *ambiguous* models, henceforth referred to as *imprecise models*. Since MOF is very similar to UML, these problems afflict both modeling and metamodeling in MDA.

Non-uniform treatment – *Uniform treatment of models* is not realized. Atkinson and Kühne [10][9] found that traditional metalanguages, like MOF and EMF, are unsuitable for *multilevel metamodeling*. They report about problems that emerge when the traditional two level object-oriented modeling is applied across the three levels of models, metamodels and metametamodels. They observe anomalies that hinder the software engineering qualities of the metamodels: *shallow instantiation*, *replication of concepts*, *ambiguous classification*, *failure to express power types* and *decreased extensibility*. This lowers *interoperability* among different modeling languages.

Limited automation – *Automation* suffers as a result of *expressiveness problems* and *non-uniform treatment*. The OMG specification for OCL [73] is limited to MOF and UML [57]. Transformations suffer also from a lack of *language independence*. Firstly, *imprecise (meta)models* make transformations less generic in the sense that they become language dependent. Secondly, OMG's QVT [71] only offers a limited set of model transformation scenarios. This causes problems with data translation [65][62]. The same *interoperability* problems are found in information systems; Atzeni, Cappellari and Bernstein [13][76] consider the failure to express and differentiate between semantically different model elements.

Analysis of the Problems

The following observations about traditional modeling architectures, like MOF and EMF, may help to identify the cause of the problems:

Lack of real-world relation – Part of the found deficiencies are due to the poor understanding of the meaning of the modeling constructs [41][16]. These *expressiveness problems* can potentially cause *ambiguous* interpretation of (meta)model elements. Traditional (meta)languages like, for example, UML and MOF only implicitly base their constructs on real-world concepts [30][44]. They seem to take a rather pragmatic commitment t*o the object-oriented domain*. Any language should be built after performing a domain and requirements analysis. We see little evidence for such analysis in the current technologies for metamodeling.

Since the goal of MOF is to do metamodeling or to represent DSLs, it may be argued that it does not provide a set of constructs, which can be used consistently in this context [45][3]. The same applies to UML were it to be used for other purposes than

the design of software systems, which is quiet imaginable considering that UML is meant to be a general-purpose language.

Lack of modeling constructs – The support of *multilevel metamodeling* is limited without *uniform treatment* of model concepts. Atkinson and Kühne [10] note that models need to provide more information than they currently offer. They [9] also show that the instantiation is not only *linguistic*, but also *ontological*. The ontological instanceOf relation is only implicitly known via the semantics of the modeling language. This relation's dual role may cause ambiguity [18]. Yet in traditional modeling architectures, the instanceOf concept only plays a secondary role.

In previous versions of MOF, the M0 layer was defined for user data models. MOF did not provide a *language independent structure* for M0, which made the interpretation of this layer implicitly dependent on the modeling language [57][65]. Now that the M0 layer is assumed to be the real world and the user data models are incorporated in the M1 layer, the structure is there. Nevertheless, from the point of view of the metalanguage there is no definition of what the relation is between models and model elements in M1. For example, the UML language can tell us which `Object` is an *instanceOf*, which `Class`. MOF, however, cannot provide us with this information, since it is oblivious to the instantiation semantics of UML (or in the general case, any other modeling language).

Lack of language semantics – The domain of metamodeling is poorly understood [63][29]. Both meta and modeling languages are considered to be only *structural definitions* in current MDE practices. Thereby they limit their semantics to the model at the layer below (M2 for metalanguages and M1 for modeling languages). Atkinson and Kühne [7][3] dubbed this s*hallow instantiation*. To support multilevel modeling and automation in modeling a semantics description of the instantiation relation needs to be provided [29][82]. This is needed for two reasons: (1) in every architecture, always two instantiation mechanisms are at work and (2) automated handling of model elements requires information on their semantics:

(1) The metalanguage assumes one instantiation mechanism. This is implicitly used to instantiate models from metamodels. Nevertheless, the mechanism may be different from the mechanism that the modeling languages themselves use [8][3][5]. There is no precise definition about how these two instantiation mechanisms cooperate [21][62]. In this situation, MDE tools still need to hardcode the instantiation mechanism from metalanguage and the modeling language separately. At least for UML and MOF this instantiation mechanism is similar, but other DSLs expressed in MOF might specify different semantics there [24]. For example, OWL [90] allows an instance to be instantiated from multiple defining elements, while UML allows only one.

(2) To support *automation,* MDE tools have to implement the *instantiation mechanism* of the languages. During a transformation, a model can be updated and during querying, navigation on the model takes place according to the *instanceOf*

semantics of both the metalanguage and the modeling language. These are different for each language as we just saw. Even though the instanceOf semantics is language dependent, metamodel definition in the traditional modeling architecture, do not include semantics for instantiation.

On the base of the reported problems, we can conclude that the nature of metamodeling is not yet well understood. The result is an inadequate set of modeling constructs that cannot support a sound interpretation of models and the relations among them (instanceOf relations).

1.3 Research Questions

To improve the applicability of MDE we need a better metamodeling foundation. This leads us to ask the following questions:

***RQ 1**: What can we use as a solution domain for metamodeling?*

A metalanguage can be seen as a generic, domain-independent language but to some extent, its task is also "domain-specific": to define metamodels. A metalanguage should be built after performing a domain and requirements analysis for metamodeling. An answer to that question should provide us with a domain that includes knowledge about the nature of metamodeling. The domain should be general enough to include aspects of modeling languages, while at the same time provide such concepts with an unambiguous grounding in the ***real world***. After an approach is chosen that answers the former question, we have to ask ourselves how the new concepts can be expressed inside modeling architecture.

***RQ 2**: How to express instantiation uniformly in a modeling architecture?*

For precise and adequate metamodeling and modeling, models need to capture *the nature of the instanceOf relation* as a construct. This enables support for that multilevel modeling that we saw in the problem of *non-uniform treatment*. Only offering additional constructs does not bring a solution that also solves the problem of *limited automation*. MDE tools need to be able to use the instantiation semantics that are assumed by the different metamodels (including the metametamodel). Therefore, we need a conceptual description of these semantics. Additional language constructs can provide independent specification of the ***instantiation semantics***.

1.4 Research Objectives

The following research objectives are formulated on the base of the research questions:

1. To create a modeling architecture that captures the nature of instanceOf in models and lift metamodels from structural definitions to the level of semantic descriptions. At the same time, *precise model definition* needs to be supported. This requires:
    a) to choose an appropriate domain for RQ 1 and study its concepts
    b) to propose a modeling architecture that represents models in accordance with the answer provided to RQ2
    c) to propose a metalanguage based on the domain chosen for RQ1 that includes means to capture instantiation semantics of modeling languages

2. To improve the pragmatics of modeling and metamodeling by:
    a) to provide tool support for performing:
        1. language definition
        2. model definition, import and export
        3. verification of model and language conformance
    b) to create a model query language to demonstrate the language independence of the modeling architecture and the tools
    c) performing case studies to validate the proposed metalanguage and tools
        1. expressing UML while focusing on the instantiation of complicated constructs like association
        2. expressing MOF to demonstrate support of multiple instantiation from model elements

1.5 Approach

Here we outline the steps we take in order to answer the research questions.

RQ1:

To select a proper solution domain, *we will study the domain of metamodeling*. We saw that traditional metalanguage approaches make a pragmatic choice for the domain they draw concepts from to represent metamodels. Often the object-oriented domain is chosen. Recent work on the question draws concepts from Formal Ontology [43][90].

Ontology is the study of existence, of all the kinds of entities—abstract and concrete—that make up the world [89]. In Formal Ontology, these found categories are related to each other in a formal way. Therefore, it is a suitable candidate to base modeling concepts on. In a situation where the metalanguage makes an explicit commitment to Ontology, the languages expressed in it are forced to use these constructs more consistently resulting in *precise metamodel definitions*. The solid groundwork in the field of Formal Ontology can help to come up with guidelines to apply the modeling concepts and verify models for correctness [30][45]. *Therefore, we propose Formal Ontology for the constructs in the proposed metalanguage.*

RQ2:

In order to apply Ontology for the metalanguage, *we will reflect on our study Formal Ontology and metamodeling*. We will summarize the knowledge drawn from these domains and accordingly propose language constructs for the new metalanguage. Thereby the metalanguage will support *precise metamodel definition*. We especially concentrate on finding the concepts in the domains that are needed to support the identified *lack of semantics and constructs*. Thus, a special focus will lie on finding an ontological meaning for *instantiation relations* so it can be explicitly incorporated in the metalanguage.

The study on the domain of metamodeling will provide ideas on how to integrate new constructs and semantics in proposed metalanguage and modeling architecture [11][3][7] [37][82]. After the conceptual approach has been established we are interested in how it performs as modeling architecture. In order to verify the appropriateness of the

metalanguage for both metamodeling and modeling, we will conduct case studies. Existing modeling languages and models can be expressed in our metalanguage. UML, MOF and OWL are candidates to work with, but also data representation languages can be metamodeled [24].

To conduct case studies we need tools; thus, we will implement prototypes for the metalanguage and supporting modeling tools. This *prototype modeling architecture* will require tooling for: *language and model input and output, a model conformance checker and a query engine*. The latter can demonstrate language independent modeling. W*e chose to provide a model query tool on the base of OCL.*

1.6 Contributions

This thesis makes the following contributions:

1. *A study of the problem domain of metamodeling and the solution domain Ontology*

The problem domain of MDE has been thoroughly researched. As a solution domain, Formal Ontology has been chosen. A thorough study of this domain would be an infeasible task for this thesis. We, however, present here pragmatic research of the domain with an emphasis on those concepts that have been used for our solution. Concrete knowledge was found in the solution domain that supported our approach. We found ontological knowledge about relations, generalization/specialization and the ontological nature of languages.

A goal of this work is to propose a solution for metamodeling as a whole; therefore, we considered all the problems, looked at their causes and toke an integrated approach of applying Formal Ontology and integrating instantiation semantics. The result can be seen in the proposed metalanguage, whose constructs are based on Ontology and enable the definition of instantiation semantics. Thereby we demonstrate the application of the results of both our studies of metamodeling and Formal Ontology.

2. *An Ontology Grounded MetaLanguage capable of expressing modeling languages together with their instantiation and related generalization/specialization semantics*

We have *extended*, *concretized* and *provided interpretation* on an existing idea for this metalanguage [63]. Chapter 4 presents OGML, which draws its constructs from the domain of Ontology. The modeling languages that are expressed in it therefore automatically make their ontological view of the world explicit and are inclined to obey ontological laws. This improves interoperability between these languages.

Furthermore, OGML is capable of expressing the instanceOf relation between languages and models and between models. Because of this explicitness, OGML can provide *uniform handling of models*. OGML also explicitly provides a *uniform representation of the model structure*.

3. *Case studies to demonstrate the use of the language and its benefits*

We expressed two flavors of UML**:** *SimpleUML1* and *SimpleUML2*. While the two do not differ much in the constructs they provide (both focus on attributes and associations) their intention is different.

SimpleUML1 **–** The purpose of *SimpleUML1* is to show our meta-language can handle different structures to express "Object diagrams" with. The UML specification [75] explains (rather vaguely) how associations are instantiated to links and attributes to slots. In *SimpleUML1,* we define three different definitions for the instantiation semantics: associations instantiated to links, navigatable associations with attributes (association classes) and association instantiated to slots.

SimpleUML2 **–** focuses on the expression of **n-ary** associations [36]. We show that the result is a navigatable model. Both adaptations of UML provided us with insides in the ontological nature of the UML constructs. Their differences make explicit the design choices that have to be made when designing a modeling language.

4. *Based on the metalanguage, language independent model querying is proven to work in a prototype with an OCL implementation*

Because our metalanguage makes instantiation explicit, we can navigate metamodels and models in a *uniform way*. We have proven this with an implementation of the OCL language [73].

From the point of view of the metalanguage, every language and model can be queried against a general structure. Every language that is expressed in our metalanguage makes an additional ontological commitment, which can be queried accordingly. At the same time, the metalanguage itself provides an ontological commitment for the modeling languages. We call the different points of view here the language axis and provide an OCL implementation that can query models over these axes, with only few changes to the OCL language.

The language independence has some favorable consequences for *automation* in MDE. Potentially it can help support an increasing set of model transformation scenarios [65][62]. In a related field, it can potentially provide solutions to data translation problems, which are already being tackled with MDE techniques [13].

1.7 Thesis outline

Figure 1-1 shows the organization of this thesis. Although the structure is linear, some relations are prevailing and illustrated in the figure with dotted lines.

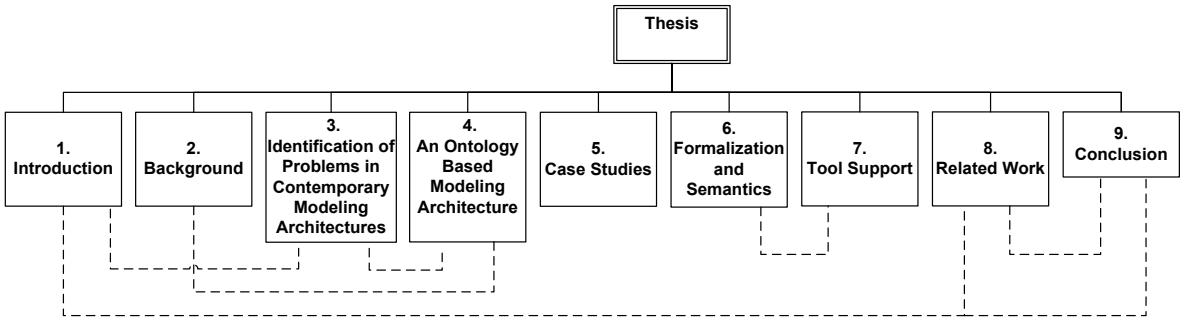


Figure 1-1 - Outline for this thesis

Chapter 1 "Introduction", the current chapter, gives an overview of this thesis. The problem description, approach and contribution sections here touch upon related works, which are further expanded in the Chapter 8 "Related Work", as you can see from the drawn relations in the outline figure.

Chapter 2 "Background" explains the relevant knowledge domains. First Ontology and languages are explained. The more detailed ventures into Ontology and linguistics are specially chosen to support the material in Chapter 4. The chapter continues with detailed explanations of modeling concepts and their semantics. We end this chapter with the definitions of terms that we assume. Because some important concepts in modeling are overloaded with meaning, we were forced to choose a particular meaning for them. A reader, who is only interested in learning about modeling and Ontology, could focus on the current chapter and Chapter 3.

Chapter 3 "Identification of Problems in Contemporary Modeling Architectures" outlines the problems of traditional metamodeling approaches. A short analysis shows the dual interpretation of the instanceOf relation in the whole modeling architecture. A conclusion is drawn that Ontology can support reasoning for (meta)modeling practices and that the instantiation semantics lacks definition in traditional metalanguages. Chapter 1 already gave an overview of the problems. Therefore, it is related to Chapter 3.

In Chapter 4 "An Ontology-Based Modeling Architecture", we present our approach to metamodeling. We introduce here a metalanguage that draws its constructs from the field of Ontology and has an explicit notion of the instantiation relation that is uniform over all modeling layers (although we have to give a different interpretation to the modeling layers within this modeling architecture). This chapter extensively uses the information about ontology and modeling presented in Chapter 2 and Chapter 3. Therefore, a reader is strongly encouraged to read the preceding chapters first unless he is already an expert in those domains. The end of Chapter 4 explains the OGML modeling architecture.

The metalanguage of Chapter 4 is then applied in Chapter 5 "Case Studies", where we use it to express existing languages. Several variants of UML are expressed in the metalanguage to demonstrate its use and capabilities. The case study at the same time shows how ontological reasoning can support metamodeling decisions. In addition, the differences make explicit how the ontological nature of language and model constructs changes depending on the interpretation of the UML specification.

Chapter 6 “Formalization and Semantics”, gives a formal basis for the metalanguage, which is needed to prove correctness of the self-reflective nature of it. The semantics presented here can be used realizing tool support, which is done in the subsequent chapter.

Chapter 7 “Tool Support” presents a design for a tool suitable to perform metamodeling and modeling practices with our metalanguage. The current implementation is also presented here. Even though it is not yet integrated into the development environment, it already provides the elementary modeling tools: the syntaxes to express language and models, a type-safe OCL interpreter and a model conformance checker

In Chapter 8 “Related Work”, we compare our work with other works. Although in this introduction, we already referred to several related works. Chapter 8 explains them in more detail as well as advice for improvements to the metalanguage. The chapter concludes with an evaluation of our contribution.

Chapter 9 “Conclusion” *e*stablishes a conclusion on our achievements. The research questions and objectives of the current chapter are used to establish to what extent we reached our goals. The results are compared with related work, thereby defining again the concrete contribution. We conclude here with “Future Work” that gives some ideas for continuation of this project.

Chapter 2 – Background

2.1 Introduction

In the current chapter, we introduce the basic concepts used in this thesis. In Section 2.2, we start with the domain of *Ontology*, because it is relatively independent from other fields and can therefore be explained stand-alone. Ontology provides us with ontologies: categorizations of the world. A choice is made for a specific ontology: Four-category ontology. In Section 2.3, we discuss the concept of *language*. Several relevant aspects of computational linguistics are explained and some definitions are used. Among them, the most important is *ontological commitment,* which will be introduced before in the section about Ontology. *Modeling* is explained in the section 2.4, where we give extra focus to concepts that are frequently used throughout this thesis. Especially for modeling, we give definitions for concepts where their use in this thesis requires a new or specific meaning.

2.2 Ontology

2.2.1 A Short History of Ontology

Οὐσία (Ontia or Ousia) is ancient Greek for *"meaning"*. It bears a relation to the Latin words *essentia* and *substantia*, which are the equivalent to *substance* and *essence* in English. In the western part of the world, it were the Greeks that started with building a philosophy to describe the ultimate essence of things. Even before Socrates, people were trying to find an everlasting structure in the world, notably Heraclitus, a champion of thinking in terms of impermanence. They did this to satisfy a hunger for permanence in a world, which they perceived as always changing. In terms of Plato: to arrive to the divine realm from a worldly realm [79].

Logical theories developed through history and with it, Ontology did. Aristotle's Ontology can be traced back via Porphyry, the Scholastics, Lull, Leibniz, Boole, Peirce, Frege, Schröder, Peano and Russell [89]. The specific scientific discipline of reasoning on Ontology and making explicit its assumption in logic is often called Formal Ontology [40].

2.2.2 Ontology

Sowa defines Ontology as:

> *"Ontology is the study of existence, of all the kinds of entities – abstract and concrete – that make up the world"* [85].

It tries to classify, make categories (ontologies), of the things we perceive using only two sources: perception and reasoning. Ontology is therefore based on one end on the findings in cognitive science and on the other on logics, which can be used to derive new facts from former conclusions. The study of Ontology, with capital O, is concerned with finding a general ontology for things in the world whereas ontology, small o, can be a more specific categorization [40]. Biology for example provides an ontology to categorize (classify) organisms: species, genus, family, order, etc.

Ontology recognizes that humans perceive the things in the real world through the different models (or ontologies) we have of it [90]. If we are interested in cooking something to satisfy our appetite, we look at the spices in the shop and consider the property called taste. If we are however interested in commerce and want to trade the spices, we make a model of spices concerning the properties: land of origin and shelf date (the latter might also be of concern for less opportunistic cooks). Every instant of our daily life we make ontological commitments like that, depending on our motivation and goals (whether we know them or not). If we zoom out, we see that the commitments are defined culturally as Guizzardi showed in his thesis by referring to work from the field of the cognitive sciences and especially from anthropology [45][1].

Ontology versus ontology

We have seen that we have Ontology and specific ontologies. Ontology is the scientific discipline of defining categories and finding relations between them. An ontology can be a general term. In Artificial Intelligence, any concrete representation of a specific reality can be an ontology. According to this view, a classification of things we find in, for example, companies is an ontology. A more narrow interpretation of ontology would be the different (world) categorizations resulting from the study of Ontology. We will use the term only in the latter sense in this thesis.

Different Views on Ontology

Mylopoulos [68] proposed to classify ontologies into four categories: *static*, *dynamic*, *intentional* and *social*. Each of these categories focuses on different concepts in the real world. In this thesis, we are mainly interested in *static* ontologies. We do not want to exclude the metamodeling of process-oriented languages and we do not have to, because their models can be considered static representations of processes.

A central discussion in Ontology is whether a "concept" should be counted as a category on its own. Some views do, and often call it a universal, while others do not. There exist more of these central discussions within the community of philosophers. The discussions have resulted in several static ontologies. Here we present two: Bunge-Webber-Wand and *Four-category ontology*.

2.2.3 Bunge- Webber- Wand Ontology

The Bunge-Webber-Wand (BWW) ontology does not distinguish *concepts* from individuals and therefore has no category for concepts of thought. It takes the view that universals (called *Kinds* in BWW) are established *a posteriori* from the presence of sets of properties in objects. BWW is however frequently used in the information systems community for several reasons [30]:

- It is well formalized in terms of set theory and has not been developed specifically for use in information systems analysis and design,

[1] An interesting non-scientific source is "Blackfoot Physics: A Journey into the Native American Worldview", where the process-oriented world of thought of some Native Americans tribes is explored, whose language is verb-oriented and their science experience-oriented.

- It has been successfully adapted to information systems modeling and shown to provide a good benchmark for the evaluation of modeling languages and methods,
- It has been used to suggest an ontological meaning to object concepts,
- It has been empirically shown to lead to useful outcomes.

BWW recognizes three main categories: Thing, Property and Law. A combination of these, results in the derived categories: *Kind* and *Attribute*. A *door* and a *pan*, for example, can be of kind *object-with-handle*. A *law* can be used to specify relations between properties of kinds. Once a property is observed, it is called an attribute. This is discussed in a following subsection.

2.2.4 Four- Category Ontology

Four-category ontology (FCO) is also used in several contemporary works on the use of Ontology in modeling [45][26]. In FCO, the basic distinction is between *individuals* and *universals* as the most fundamental entities of being. FCO thus recognizes classes *a priori*. The ontological study that claims the existence of universals is known as *metaphysical realism* [2][66]. The second division is between *substantials* and *non-substantials* (moments) based on the notion of independent existence. For example, the color property that things may have is not substantial. The two divisions are orthogonal, thus resulting in four categories [23]. Figure 2-1 depicts the concepts in this ontology.

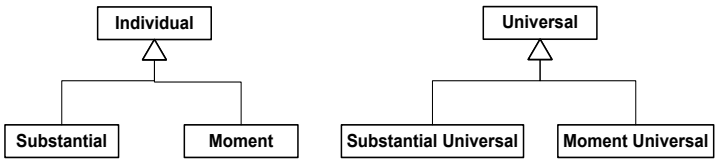


Figure 2-1 - Four-category ontology (taken from [63])

Individuals are classified as *Substantial* and *Moment* individuals. A substantial individual or just *substance* is something that can exist by itself without depending on the existence of other individuals. This existential independence is the core feature of substances and gives the major distinction from moment individuals. Examples of substantial individuals are cars, people, books, etc. In the programming languages and modeling languages, substantial individuals are usually represented as objects (e.g. Java object and UML object).

Moments are individuals that exist in other individuals. Moments cannot exist standalone, they are existentially dependent on at least one individual (called *bearer*). Example of a moment is the red color property of a car. In that case, the red color moment exists in the substance car. The relation between a moment and its bearer(s) is called *Inherence* relation. Moments may inhere in more than one individual. In programming and modeling languages, moments are called in various ways: *slot* and *link* in UML, *field* in Java, etc.

Universals are entities that can be instantiated in individuals. According to Aristotle, universals can only exist via their individuals and not independent from them. The individuals that exemplify a universal have something in common. For example, things that consist of matter have mass. The actual value of the mass varies but the mass is observed as a common property of individuals. In this case mass is a universal.

Universals are classified into *substantial universals* and *moment universals*. As the names suggest, substantial universals are exemplified (i.e. exist through) by substantial individuals and moment universals are exemplified by moment individuals. *Instantiation relation* is the relation between an individual and a universal that exists in this individual. Universals have their representatives in the existing computer languages. UML classes correspond to substantial universals. UML attributes and associations correspond to moment universals.

This choice of categories results in an ontological square represented in Figure 2-2. Normally *characterization* is the term used for both the relations at the universal level and at the individual level. For distinguishing them, we will use the term *inherence* on the individual level.

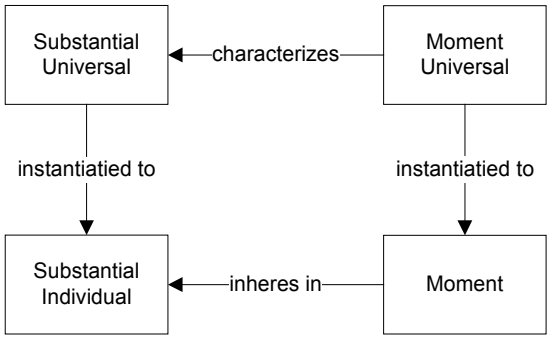


Figure 2-2 - The ontological square

Properties and Relations in Ontology

Ontology recognizes the existence of *relations* between (among) universals and also between individuals [67]. Commonly they are represented as *"properties"*, therefore we will use it here also in favor of "relations". We resort to the Formal Ontology of BWW to derive knowledge for properties. It is useful to mention here that these results are not less applicable within the context of FCO, because we can map properties to moments and individuals to things [43].

Properties of Properties

Webber et al. [90] talk about "attributes" ("substantial properties" or "predicates") of universals and properties of individuals. They were able to make important postulates on the use and interpretation of properties, which should support the use of this concept in MDE and database systems. To start they first define properties as being always "attached to things" [90]. Properties can be attached to multiple things in which case they are *mutual* (relational) otherwise, they are *intrinsic*.

Mutual properties are relations between things. Intrinsic properties can have "values" which are not other substantials or individuals in the model. *"Values are elements of the codomains of attribute functions. They cannot exist independently in the world. Instead, they must be conceived in terms of things that have properties that in turn are represented as values of attribute functions"* [90]. This makes it seem like they exist inside the individual or universal.

Perspectives on Properties

Analogous to our story about spices in our introduction of Ontology, Webber et al. [90] write: *"The properties of a thing exist, whether or not humans are aware of them. Humans conceive of things, however, in terms of models of things. Such models are conceptual things. …. Attributes are characteristics assigned to (models of) things according to human perceptions. Depending upon circumstances, humans may use different models of the same thing, and therefore assign different sets of attributes to the same thing"*. This observation recognizes the need for different perspectives on the objects of our perception.

Attribute Functions

For this thesis the most important aspect regarding properties, is the seemingly simple fact noted by Webber et al. that properties are actually relations over sets of instances. The problems with representing properties in information systems stems usually from the fact that they are treated as functions. Therefore Webber et al. introduce the **attribute function**, which is a means to look on properties from the different perspectives of the instances that share them or "participate in" them.

Laws

Webber et al. give several laws (constraints) for the use of properties and substantial properties. To give a few examples: kinds (substantial universals) should have properties, properties cannot have properties and a property without value is not a property.

2.2.5 Generalization in Ontology

Ontology has a notion of generalization. Although there are different views on the issue [23] they have in common that generalization relations are expressed with laws. In an ontology with a category for universals, these laws could be of form: `universal x` is of kind `universal y`. The laws could also be described more property centric. To detail on the consequences would go far beyond the scope of this thesis. We will only use the laws that establish relations between universals.

The effect of generalization/specialization is of course that the specializing universal "gets" the properties of its "general" universal. In BWW a special case is distinguished [30]; "*the specialization of properties. That is, when a Kind possesses a property which is a specialization of a property of the general Kind*"[2]. To give an example: a `vehicle` has a property `can_move`, whereas a `plane` has a property `can_fly`. To handle specialized properties they need to make their nature explicit in the ontology.

2.2.6 An Ontological Commitment

Making an *ontological commitment* means to presume a certain ontology. Natural language does this. For example, the sentence "Napoleon is an ancestor of mine" assumes two things, the speaker and Napoleon. We can however only derive meaning from the sentence if we assume another thing: ancestors. This is a category, where all of the speaker's ancestors are included. Second-order logic is now needed to establish the truthfulness of the sentence. It

[2] Quote adapted to chosen terminology

should be understood that we have to distinguish between statements and questions. Where the statements are used to form a knowledge base, questions are used to derive knowledge from it.

2.3 Languages

2.3.1 Linguistics, Syntax and Semantics

In the current section, we explain what there is to a language. The research of languages starts with natural languages. The cognitive science that deals with research on natural languages is called *linguistics*. Linguistics deals with the *semantics* (meaning) and *grammar* (structure) of languages. Grammar can again be decomposed in morphology (formation of individual words), syntax (rules for the composition of words into sentences) and phonology (abstraction over the sounds that words are composed of) [22].

Ever since languages are used in computers for the purpose of *programming* and *data representation*, the results of linguistics have been used in this new environment. For example, Chomsky [25] took a generative approach to formalize syntactical grammars and found that only context free grammars are invertible. These results are used in compiler construction, where the syntax of a language is often represented by a *concrete syntax* in Backus Naur Form (BNF)[3] and an *abstract syntax* [88]. The latter can be represented as a model and is of interest for this text. A parser "parses" the concrete syntax of the textual definition in a language (see Figure 2-4). The result is what is called the *abstract syntax tree (AST)* that conforms to the model of the abstract syntax.

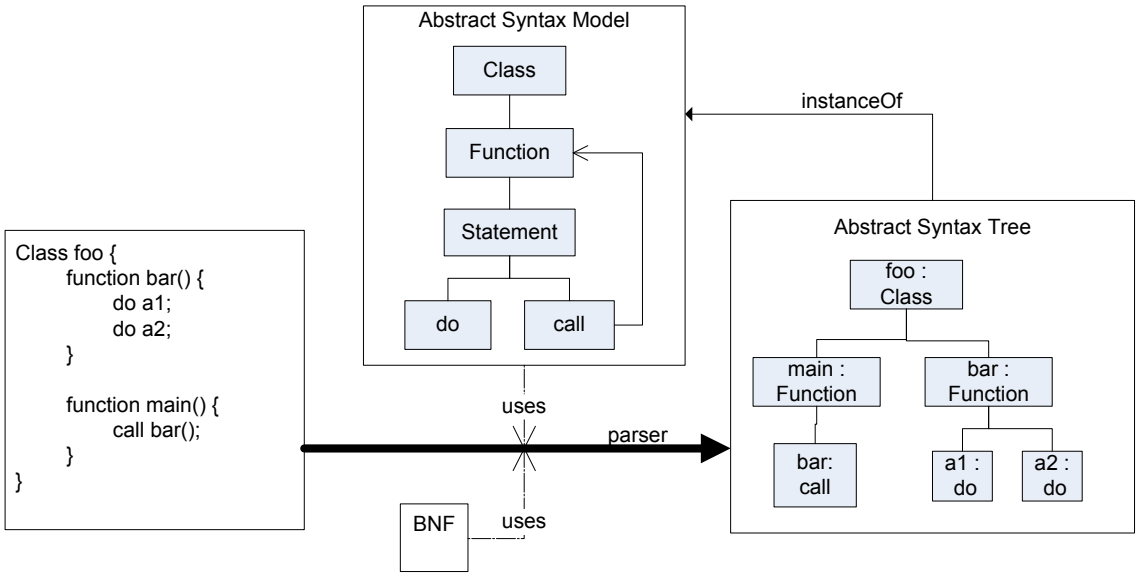


Figure 2-3 - An abstract syntax model and an abstract syntax tree

The *syntax* or *notation* of a model can just as well be graphical. Languages as UML use a graphical syntax whose main elements are boxes and lines. We call such a language

[3] See Appendix A – for an example BNF

diagrammatic Parsing of diagrams is much harder than parsing of textual syntaxes [46]. The results however are the same: an AST.

Semantics. give meaning to a language. Semantics are expressed by mapping the abstract syntax. onto some semantic domain [46] (see Figure 2-4). For a general-purpose language, this domain could consist of a set of data types like integers, lists and strings. Object-oriented languages could have a domain of objects and DSLs have specialized domains, like state machines or processes. For *data description languages* (for example RDF and XML), where the AST is often sometimes used directly in the software system.

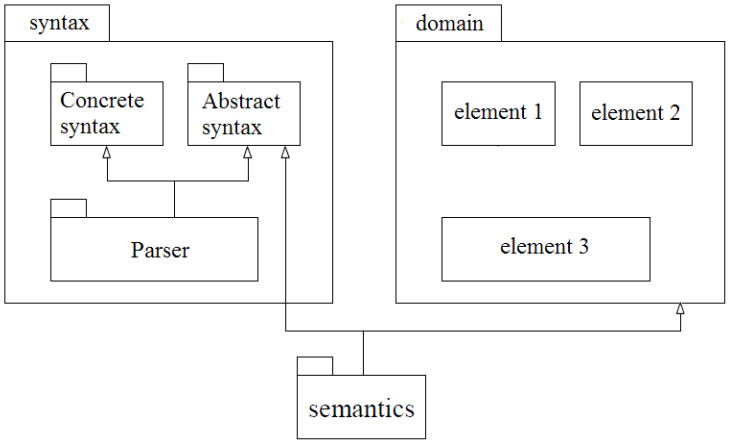


Figure 2-4 - Concrete syntax, abstract syntax and semantics

2.3.2 The Pragmatics of Modeling

For the in- and output of models languages are used. These modeling languages can be diagrammatic. (visual) or textual. In UML and MOF, diagrammatic syntaxes are dominant. Some technologies, like KM3, use textual languages for modeling. There is, however, more to modeling languages than just syntax. The next section elaborates on their semantics.

2.3.3 The Ontological Commitment of Languages

Milton and Kamierczak [67] analyzed the models of different languages and concluded that they were valid instances of Ontology[4]. Guizzardi [45] noted that this observation can be generalized under Quine's *Ontological Commitments* [77]. For this reason, the definition of languages can be viewed as making ontological commitments.

We see this also in modeling and programming languages. Java for example assumes the existence of universals and models them as classes with fields (moment universals). Runtime instances of these classes are objects with properties (moments). The same ontological commitment is made by UML where the classes exist in the Class Diagram and the Objects in the Object Diagram. The language constructs and semantics decide thus what view a language takes on the real world. Thereby it determines what facts it can express about it.

[4] The specific ontology is not relevant here.

2.3.4 Expressiveness or Precision

When we draw an explanation on a whiteboard, we implicitly use a language. This language may or may not be precisely specified somewhere and can accordingly be interpreted. The more precise the language used, the less ambiguity can arise when we interpret the model. However, sometimes it can be a powerful means to express knowledge in a informal way, for example with different arrows and boxes [45]. Natural language is also such an example. An attempt to specify its semantics and syntax is made with dictionaries, encyclopedias and grammar books; however, it is not difficult to create sentences with ambiguous meaning.

There is always a trade-off when creating a language. Either the language is less expressive and unambiguous, or the language is expressive and ambiguous. If we want a more formal definition of ambiguousness for a language, we have to relate it to the number of undecidable statements (or propositions) that can be made with it. Gödel [35] was the first mathematician to prove that any language can be used to make undecidable statements in his work called *"On formally undecidable propositions of Principia Mathematica and Related Systems"*.

A consequence of Gödel's results is the fact that expressiveness and ambiguousness are properties of inverse proportionality for each language. This fact is easy to establish in the real world; whereas it is easy to come up with any number of ambiguous statements in a natural language, it is much harder to produce and understand them for precise mathematical languages. It took quite some time for mathematicians to first come up with the Russell paradox and later solve it. From the invention and formalization of set theory around 1800 [86] to Russell in 1902 [80] and from 1902 to 1931, when Gödel solved the problem.

When defining languages thus we have to keep two things always in mind: stay away from completely defining the semantics in the language itself and be aware of the disagreement of a domain-specific design and a general one. This therefore applies to metamodeling practices and in even greater degree to metalanguage design. It also provides a technical reason for the necessity of DSLs. In the next section, we show some languages in MDE and discuss how they define themselves.

2.4 MDE

2.4.1 Model Driven Architecture and Engineering

Software systems grow larger every day, while at the same time growing more dependent on other (software) systems: the ones they cooperate with, but also the software technologies used for their implementation. This causes problems in portability, interoperability and productivity of these systems, as a great deal of time has to be invested into activities like low-level design and coding. These are difficult and error-prone processes.

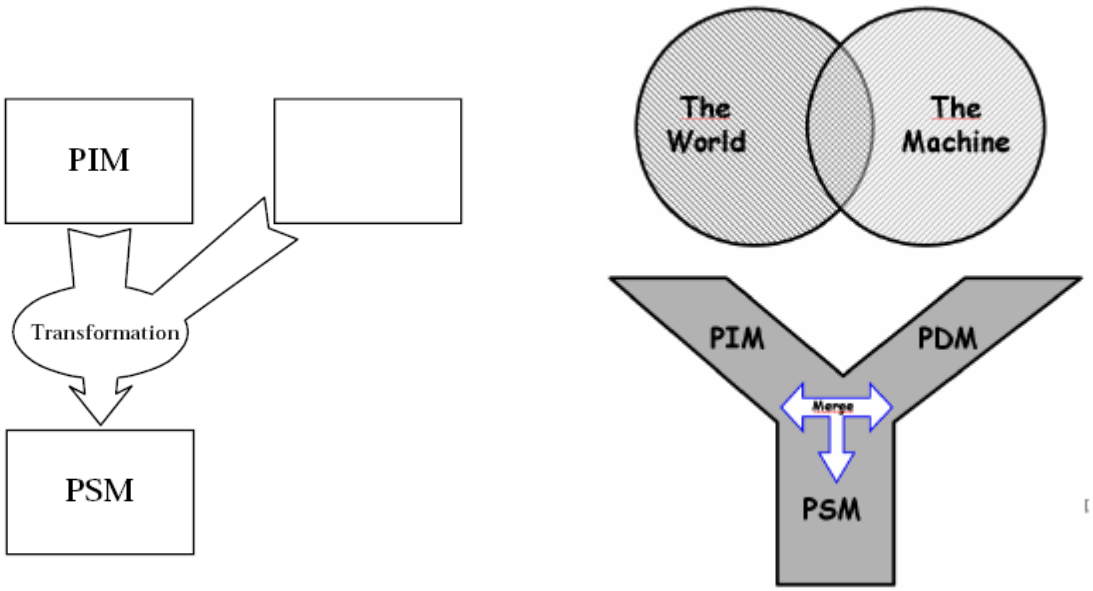
Model Driven Architecture

The Object Management Group (OMG), a consortium of software industry participants, proposes MDA as a solution for these problems [70][56]. MDA promotes the use of models in

software development. It raises the level of abstraction in software development by treating models as primary artifacts and emphasizes the activity of *modeling*.

Two basic principles are applied in engineering disciplines, this also the case in computer science. First is the use of *models* to express knowledge about the design of software systems. Second is the separation of system specification from its implementation and technology specific details. Therefore, MDA defines two classes of models[5]: *Platform Independent Models* (PIM) for system specifications, which are independent of the platforms they can be implemented on, and *Platform Specific Models* (PSM), which describe the system with the details of a specific implementation platform. The abstract layer can be the conceptualization of objects that exists in a system in UML, and the more concrete layer can be a programming language, which represents these conceptualizations, for example classes in Java.

The development of a system according to MDA starts with the definition of a PIM. This PIM can then be transformed into a PSM using additional knowledge about the platform. This process is called model transformation and depicted in Figure 2-5. OMG provides a standard for model *transformation*, called Query / View / Transformations [71] . It relies in turn on the standardized model query language OCL [73].



a) transformation in MDA (taken from [76])

b) a depiction of the proces including the notion of Platform Dependent Model (PDM) (taken from [19])

Figure 2-5 - A model transformation between a PIM and a PSM

Model Driven Engineering

Model Driven Engineering (MDE) describes how MDA is used in the broader context of the software development process [55]. It includes more technologies than OMG does with MDA; XML and database systems are but examples.

[5] A third class is recognized in the requirements phase: the Computational Independent Model (CIM). We do not include it here.

Furthermore, MDE adds different dimensions to modeling. The development process is concerned with more properties of the system under development than only its structural design. Different concerns can be security, distribution and error handling. All these things could be modeled orthogonal to the structural design. In addition, most software development processes specify an incremental process, which results in stack of versioned artifacts [58][14]. Some argue that the need for model versioning needs to be handled by making models of modeling architectures [19], making MDE an even broader field.

2.4.2 The Concept of Model

Various sources can be consulted to establish a meaning for the word "model". It is a central concept within MDE and often used in this thesis. To establish a useful definition we look at how it can be interpreted from the point of view of different sciences. First, we take a look at in what context we use models. This subsection concludes with a choice of terminology.

Context of Use

Models can be expressed in a precise modeling language or with an ad hoc notation to give a quick explanation on a whiteboard. Whereas the latter option provides more expression power it can easily be the cause of ambiguous interpretations [45]. In computer science, we therefore need a more formal representation of models. The ability to express the meaning of a system unambiguously can not only prevent misunderstandings in communication but also even be the basis for automation in the software development process as we have seen in the previous section.

Semiotics

From the point of view of semiotics, the study of signs, their syntax, semantics and pragmatics, the denotational aspect of models is emphasized: a model defines a set of symbols for notational purposes. These symbols are related to the entities in the real world that they represent and to the concepts of these entities that exist in the mind of the model designer. Ogden and Richards are credited for recognizing this fact and later Ullmann represented it in the triangle that is depicted in Figure 2-6. The FRISCO report [52] extended this meaning triangle with an actor that makes an interpretation of all three corners.

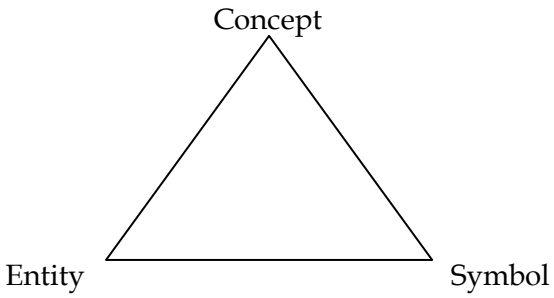


Figure 2-6 - The meaning triangle adapted (taken from [62])

Kurtev looked at existing definitions for models and came up with the following definition, where the word "object system" is used to describe the part of reality that the model expresses:

> *"A model represents a part of reality and is expressed in a modeling language. A model provides knowledge for a certain purpose that can be interpreted in terms of the object system"* [62]

Conceptual modeling

Guizzardi [45] created the image in Figure 2-7 to explain his view on models. His definition focuses on the use of models in conceptual modeling. In his view, the real models reside in the real world. Modelers compose specifications of them. The picture also makes explicit the dual relation that models bear towards: (1) the real world and (2) their conceptualization expressed in a modeling language. However, unlike Guizzardi, we are not concerned with conceptual modeling. Thus, the definitions he uses, which emphasize the informal aspect of modeling by including understanding and communication as a purpose of models, does not directly apply in our situation, where we want to apply modeling in MDE.

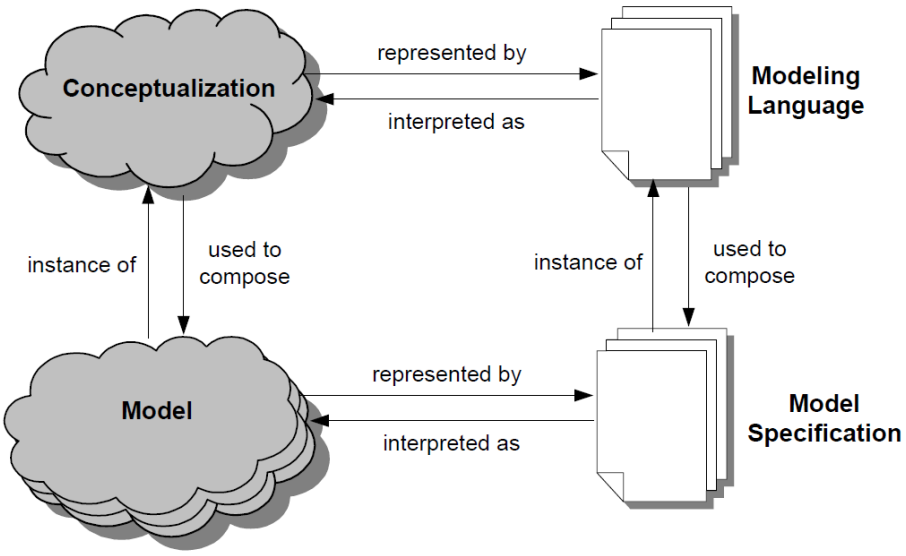


Figure 2-7 - Conceptual modeling according to Guizzardi (taken from [45])

Formalisms

From a more formal point of view, we have to recognize that models represent an *"abstraction of reality (real or language based system) in order to make predictions and inferences about it"* [59]. Therefore, they have to preserve a structure and thus there always exists a homomorphic relation between models and the objects in reality, from which they abstract. Kühne [59] decomposes this abstraction relation into three components: the projection function (homomorphic), another abstraction on elements (symbols) and a translation function. The latter function formally represents the fact that models are captured in a language.

Ontology

From the point of view of Ontology, the emphasis is more on the reduction aspect of models. According to Wand, Storey and Webber *"the properties of things exist, whether or not humans are aware of them. Humans conceive things in terms of models of things"* [87]. Quine calls the process of choosing a model for reality, making an *ontological commitment*. When we model a

software system, we make commitments to different ontologies, like processes and threads, data structures, objects and classes.

Assuming a Definition

According to our use of the terminology previously discussed, we are forced to establish a more concrete meaning to them. We will do this in the current section.

In subsection 2.4.2 we reviewed the meaning of the term "model" according to different sciences. We are now looking for a fundamental foundation for the term with a focus on capturing the ontological meaning of models. We are thus not pressingly concerned with the use of models for documentation and communication purposes. The formal definition gives a good insight into the nature of the relation between models themselves, models and the real world, and models and the languages in which they are expressed. This detailed notion, however, is not of primary concern for a definition of the word "model", and can be generically captured with the word *abstraction*.

What is true for our perception of the real world is just as true for our interpretation of models. Depending on the knowledge we want to derive from them, we may concern ourselves with different properties represented in the model. In the definition of the word model that we adopt, we express this fact:

> *"A model is an abstraction of a part of reality called object system (a software system, a machine, etc) from which its user wants to derive knowledge for certain purpose(s). A model is expressed in a modeling language and can also be interpreted according to an ontological commitment compatible to the one that the knowledge domain presumes."*

The definition is based on Kurtev's, but the word "abstraction" is used instead of "representation" to emphasize that models provide a view on reality. "Abstraction" also implies that the model is a denotation of reality; this is not made explicit in the definition, since we are not primarily concerned with syntax and notations.

2.4.3 Instantiation

We distinguish *instantiation* from *generalization*. Whereas generalization can be seen as a means to represent common properties of several types in one more general type, instantiation is orthogonal to it and uses types as templates to create the more concrete instances. In the process of instantiation, concrete values are assigned to the properties of types [3]. Figure 2-8 shows the difference with an example UML Class Diagram. `AnotherType` is the specialization of `Type`, thus it "inherits" its properties. `at` is an instance of `AnotherType` and has assigned instance `yat` as a value for its property "property". This illustrates that the semantics of instantiation also depend on the semantics of generalization.

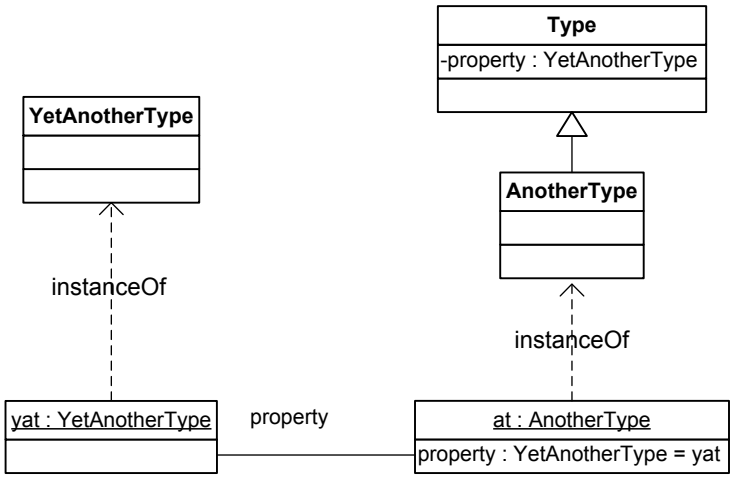


Figure 2-8 - Example of instantiation and generalization

Different Kinds of Instantiation

Figure 2-9 establishes the relation between the constructs in a modeling language and a model expressed in it. In the previous section, we already mentioned that the abstract syntax of a language can be expressed as a model. The gray part of the figure represents the abstract syntax, or; *the language constructs.* The upper part is used for class diagrams and the lower for object diagrams (this language is similar to UML). This image explains two notions of the *instantiation relation* (*instanceOf):* the *linguistic* and the *ontological instanceOf*. Both notions are, however, *relative* depending on the point of view of the observer. We will therefore introduce the terms *intension* and *extension* later.

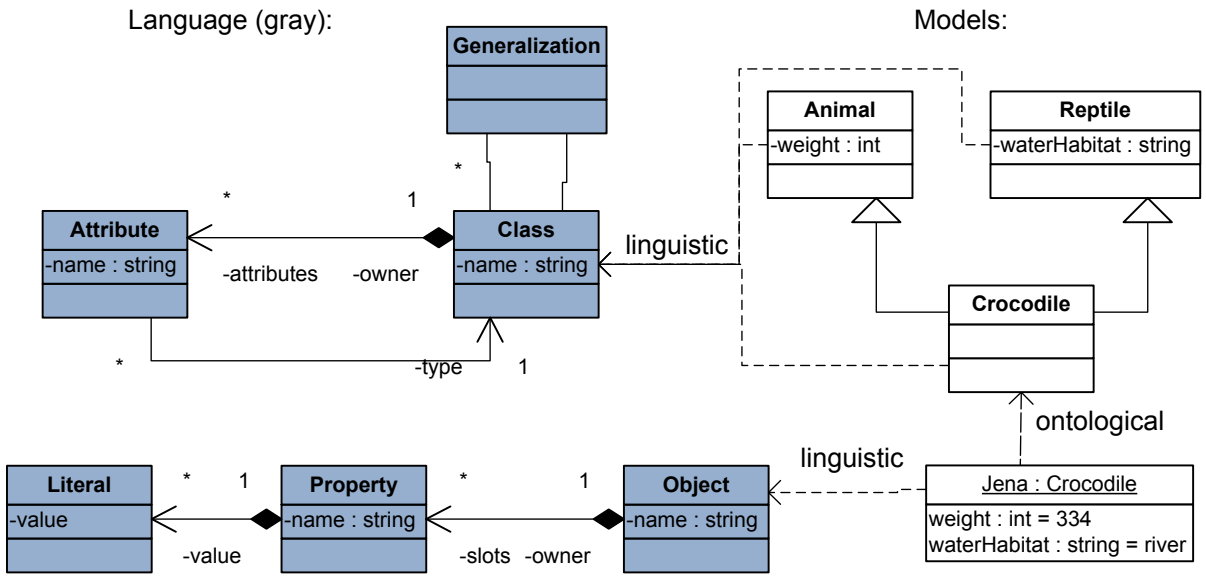


Figure 2-9 - A modeling language with two models

Linguistic Instantiation

The right part of Figure 2-9 shows two diagrams: a class diagram (upper part) and an object diagram (lower part). Both of these diagrams are "built" using the constructs from the languages (represented by the dashed arrows). This *instanceOf* relation is therefore called: *linguistic instanceOf*.

Ontological Instantiation

Figure 2-9 establishes, besides the linguistic relation between models, also another kind of relation: one of ontological nature. This relation arises when we recognize that the class diagram provides us with insights on how to interpret the object diagram. Instead of dictating the structure of the elements, as the linguistic instanceOf does, it gives a *meaning* to the elements in the object diagram [9][82]. This is shown by the dotted line from the object "Jena" to the class "Crocodile".

From this relation we can deduce that "Jena" is a crocodile: has all the properties of the crocodile and behaves like a "Crocodile" (assuming there was also a functional specification to this class). Without this relation, we could just create objects with random properties and relations to other objects. The objects would so to say "hang in the air", giving us no insights on how to interpret them. The class diagram does this.

Different Terminology for Instantiation

Bézivin [17] noted the different uses of the term "instanceOf`. Some uses overlap some do not. For a concise terminology, it is important to keep a distinction between the different meanings. The term "representation" is often used to denote a relation between a model and a system or a model construct and the real world. This relation is always one-to-one and therefore different from the instanceOf relation. It can be represented by an instanceOf, as is done in recent interpretations of the four-layered MOF modeling architecture [7], but not vice-versa.

Instantiation and Related Concepts

Figure 2-10 shows the difference between *instanceOf*, *memberOf* and *conformsTo*. With instanceOf we usually mean that the instance is a directly instantiated from a type, whereas the conformsTo relation also applies to indirect instances (via generalization). Both can be used for model constructs and models. MemberOf is often used to indicate that an instance is among those instantiated from a certain type.

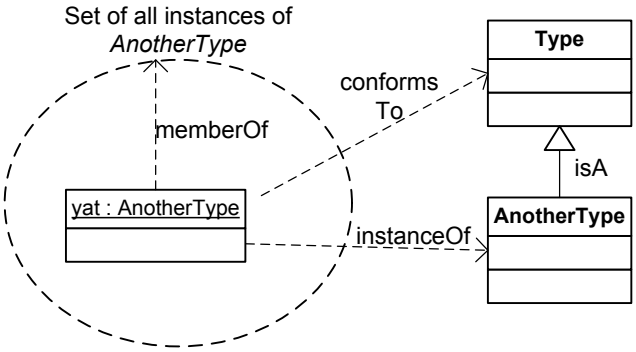


Figure 2-10 - The difference between instanceOf, conformsTo and memberOf

We distinguished two kinds of instanceOf: a linguistic and an ontological one. In other literature, these are sometimes referred to as respectively the physical and logical instanceOf [7] or instanceOf and meta-instanceOf [18]. Geisler et al. use the terms inter-level instantiation and intra-level instantiation [32]. Sometimes the word "structural" is used for linguistic instanceOf. In this thesis, we will stick with the terms linguistic and ontological instanceOf and instantiation in general. Sometimes we use "structural" to emphasize that something is "merely structural".

We distinguish also between terminology in object technology and MDE. In object-oriented technology, the term "inheritance" is used for generalization. In general the term *instanceOf* is also used for instantiation of runtime objects. In this text, we deliberately stay away from object-oriented terminology because we use Ontology as a solution domain as motivated in our approach.

We will use the words instanceOf and instantiation as inverse of each other. A construct X is an instanceOf Y and Y is instantiated to X. To process to instantiate something is called instantiation.

2.4.4 Relativity in Modeling

In the previous section, we saw that the abstract syntax of a language can be represented as a model. In modeling, this can be applied recursively; the model of one modeling language can be expressed in another modeling language. In Figure 2-11, a language stack is shown by using meaning triangles for the languages and models. The corners of the triangles are related in different ways. Between symbols and entities, a partOf relation can be found, expressed in the figure with the set partOf character. Between concepts and symbols, an instantiation can be found, expressed in the picture with an arrow. Model *m* is expressed in language L2 that in turn is expressed in language L1. The figure shows how the symbols of L2 can be interpreted relatively. From the point of view of model *m,* these symbols represent the different concepts in the model, whereas from the point of view of L1, a symbol of L2 is seen as one of its entities[6].

[6] We found a similar, although less aggressive, use of the triangle in [87]. It should be noted that the interpretation of our image could go much deeper than explained here. We keep the extra complexity, because it could give the reader a feeling about the complex nature of metamodeling [85][7].

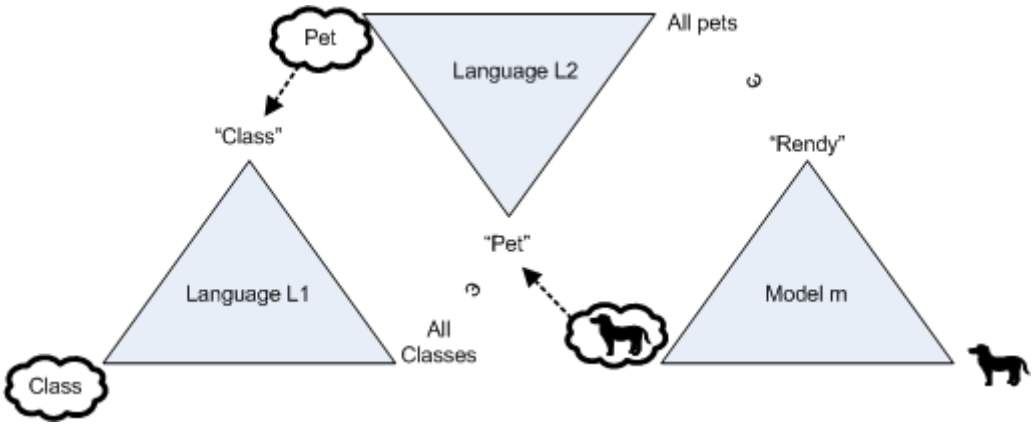


Figure 2-11 - The meta-property of models with meaning triangles[7]

We have explained the *relativism* from the point of view of semiotics. For the purpose of this text, it is also useful to understand it from an ontological point of view. Therefore, we first assign real-world values to the semiotic classes in Figure 2-11. This is done in Table 2-1. Both languages might be general-purpose as in the OMG modeling architecture, where MOF and UML are used. Here we use a DSL for language L2 specialized for the domain of pets. A general-purpose modeling language would use a concept like *Object* in the place of *Pet*.

Table 2-1 - The meaning of labels in Figure 2-11

	Language L1	**Language L2**	**Model m**
Concept	Classes of Objects	Pets	My dog
Symbol	Class	Pet	Rendy
Entity	a class	All pets in the world	This dog

From the point of view of Ontology, the language L1 provides universals, whose instances are the universals of Language *L2*. Model m contains individuals: instances of L2. The ontological boundaries between the models are thus shown vertically as the columns in Table 2-1.

Extension and Intension Dichotomy

Because the *instanceOf concept* is relative with regard to the language perspective, we have to resort to the more general notions of *extension* and *intension*. These are common notions in logic and linguistics [62]. Any word or sign has two meanings: the extension of the word refers to the set of objects it represents while the intensional meaning is domain of all the possible things for which the word can be used. A pet for example has the intensional

[7] In the image:
- clouds are *concepts*,
- quoted words are *symbols*,
- pictures or natural language represents *entities*,
- Set inclusion between *symbols* and *entities* is drawn with memberOf (*∈*),
- The linguistic instanceOf relation between *concepts* and *symbols* is drawn by dashed arrows.

meaning of an animal kept by humans for companionship or as a household animal. The extension of "pet" is the domain of all such animals [8].

Language L2 can be used for representing pets and therefore has the intension and extension just described. Model *m* contains dogs, which – in this case – are pets; therefore, the elements of model *m* are part of the extension of *L2.* In a similar manner, the constructs of *L2* are part of the extension of *L1.* To use the terms linguistic instanceOf and ontological instanceOf here, would result in ambiguous use of the adjectives, because what is a linguistic instanceOf L1 is an ontological instance from L2 perspective.

2.4.5 The Concept of Metamodel

In the previous subsection, we showed the meta-ness of the concept model. According to our view, a model represents not merely an ontological commitment on (a part of) reality, but can also be interpreted itself via different ontological commitments, one of which is the *modeling language* that it is expressed in. A modeling language can be represented by its abstract syntax as we saw in Section 2.3. MDA takes this approach when capturing the structure of a modeling language, its *abstract syntax*. This structure captured in a model is called a metamodel. Figure 2-12 shows a schematic view of this with two models (the planes). The boxes are model constructs and the arrows represent instantiation. Between the constructs, a model can represent relations (not shown in the figure). In Figure 2-9, we already gave an example.

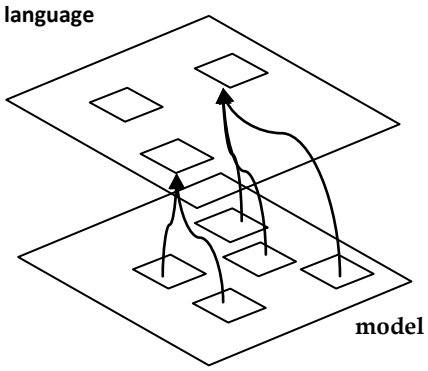


Figure 2-12 - A model expressed in a modeling language

According to this practice, we can adopt the following definition for a metamodel from the FRISCO report [31]:

> *"A metamodel is a model of the modeling language"*

One might be tempted to use the term metamodel more generally. The MDA Guide [70] for example defines a metamodel as "a model of models". This would also express the fact that we can establish models for the different interpretations upon a model (like a Class Diagram in UML). However, such a definition may cause confusion over the actual nature of the instantiation relation as seen in the previous section. Therefore, we stick with the definition of a metamodel from the FRISCO report.

The task of specifying a metamodel is called *metamodeling*.

2.4.6 Modeling Languages

With the emergence of MDE, modeling languages became more widespread. To name a few:

- OWL [90], which is used for the Semantic Web and is closely related to RDF [89],
- CWM, which is specialized in data warehousing [69],
- UML, the general-purpose language proposed by OMG [75]. UML is a modeling language that focuses on different aspects of software design. It includes several modeling languages to model *behavior* as well as *structure*. UML's roots are in the information industry and its design bears resemblance to object-oriented programming paradigms. The constructs of *the class diagram language*, a structural language, include, for example: `Class`, `Association` and `Package`. When we speak about UML in this thesis, we refer to Class Diagram language.

An interesting feature of modeling languages is their ability to describe both the model for the abstract syntax as well as a model for the semantic domain: the instances or the possible ASTs. The semantics of the language describe how these two models should be mapped. This is made explicit in MML, a recent approach to describe the semantics of UML [1]. Figure 2-13 illustrates how the definition of instantiation relation between two of its instances is the semantics of the modeling language.

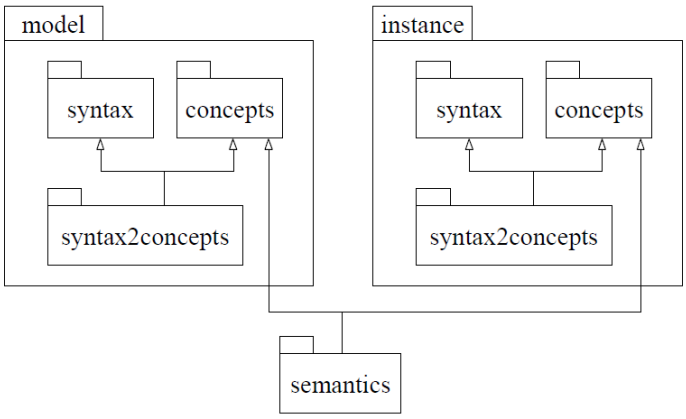


Figure 2-13 - The semantics of modeling languages (taken from [1])

In, for example, UML the object model contains constructs for the instances: `Object`, `Link` and `Slot`. In its specification, we find how the constructs are mapped: "*The purpose of a class is to specify a classification of objects and to specify the features that characterize the structure and behavior of those objects*" and "*An association declares that there can be links between instances of the associated types*". We observe that in some cases a modeling language has semantics that describe the instantiation between two of its direct instances.

2.4.7 Modeling Architectures

The structure of a model is expressed in a modeling language, which also defines the models relation to its intension (for example a class diagram). The modeling language has a metamodel, expressed in a metalanguage that has a metametamodel. The model, metamodel

and the metametamodel form the modeling hierarchy (or architecture) commonly found in MDE technologies.

MOF

UML was first defined a general-purpose modeling language and it used a set of its own constructs to define itself. At the same time, different modeling languages have been developed for different domains; RDF, OWL for the vast variety of domains on the web, BPEL for business processes, etc. Therefore, there was a need to handle them uniformly in tools. Since languages can be treated as models themselves (their abstract syntax is a model; the metamodel), different attempts have been made to create a superstructure (also a model) to express them on. MOF [71] is a pragmatic attempt. It uses the set that UML used for self-definition to define itself and other modeling languages [18].

Figure 2-14 shows the MOF linear architecture. The architecture specifies four layers of models. The architecture is strictly linear and models resided on layer M0. Strictly linear means that models only conform to the layer directly above them [36].

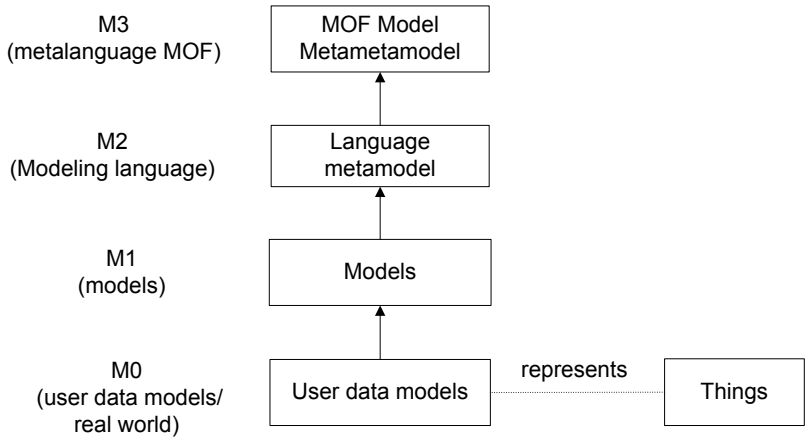


Figure 2-14 - The traditional MOF modeling architecture

Difficulties were found with the traditional interpretation of the architecture, which we discuss in the next chapter. Figure 2-15 shows a more recent interpretation of MOF in which all models reside at layer M1 and the real world is placed at the M0 layer. This is a result of the fact that the user data models are modeled in the same language as the intensional models as discussed in the previous subsection.

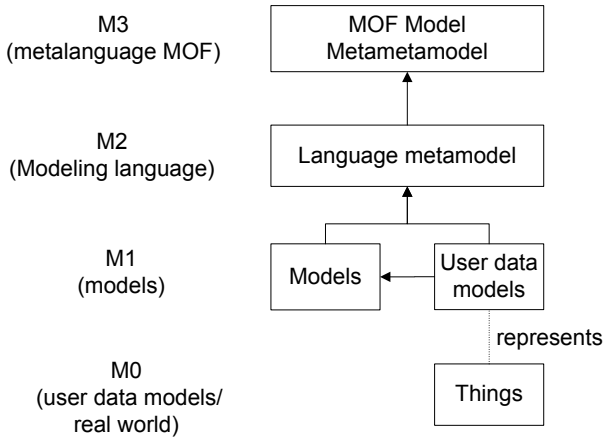


Figure 2-15 - The MOF modeling architecture, a recent interpretation

A modeling architecture should also provide means for data exchange and model input and output [9]. In MOF, this is realized by the MOF metalanguage itself. Every model is considered an instance of MOF, thus by providing a serialization mechanism for MOF; the whole architecture can be serialized [7].

The Architectures of Other Technologies

Other modeling and data description languages are self-descriptive and can be modeled [20]. Figure 2-16 shows EBNF and data description languages XML and RDF. Bowers and Delcambre [24] show how all of these languages have quite different instantiation semantics.

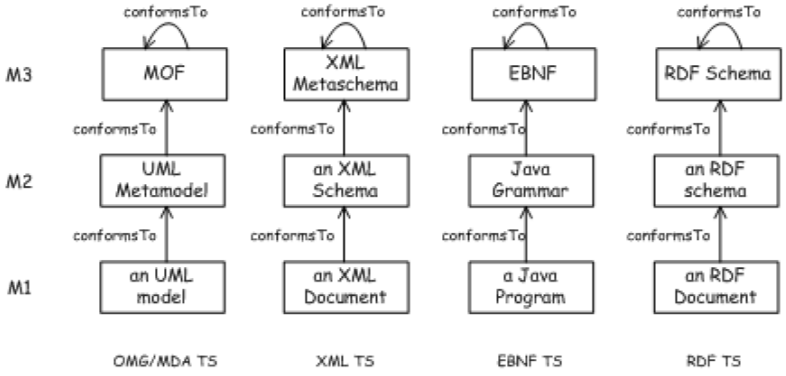


Figure 2-16 - Modeling architectures of MOF, XML, EBNF and RDF (taken from [20])

Nested Modeling Architectures

Recent approaches to create modeling architectures attempt to represent the linguistic and ontological instanceOf relation more faithfully. MML from Alvarez et al. [1] is such an example. By representing both instanceOf relations as a primary modeling constructs and expressing its semantics in a structural manner, they end up with an architecture as represented in Figure 2-17.

The basis of this new interpretation is of course seeing the metalanguage in the same light as the modeling language (as described in Subsection 2.4.5). In this light, the metalanguage also is a special kind of language whose semantics define a mapping between its direct instances:

in this case *languages* and *models*. So analogous to the way that MOF incorporated M0 into M1, in these nested architectures the new M1 is incorporated into M2.

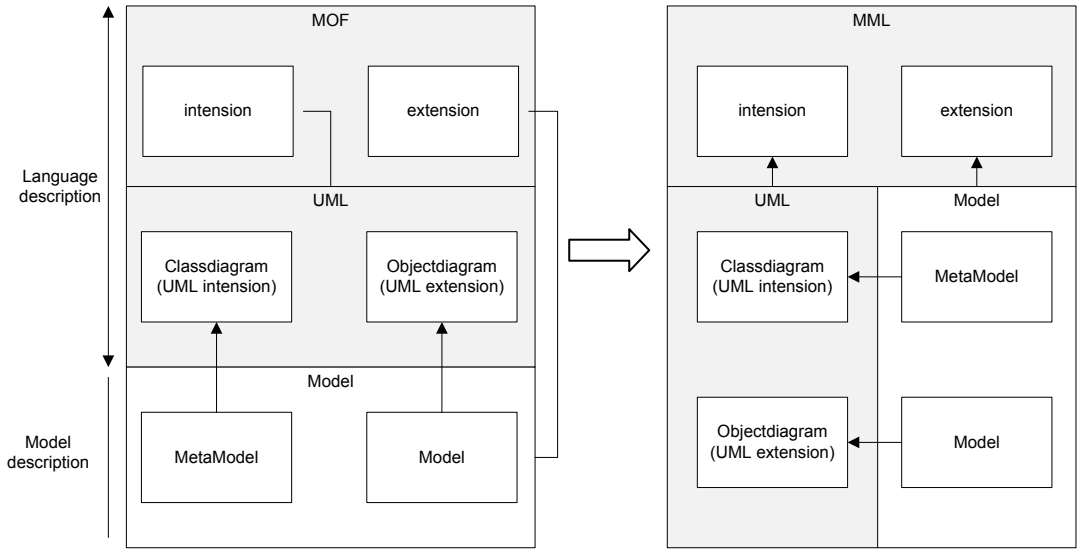


Figure 2-17 - The MML modeling architecture compared with MOF

Modeling Architectures in General

In order to evaluate different designs, Atkinson and Kühne [11] made a detailed comparison between the different options for modeling architecture design. The use different characteristics, which can be summarized by number of levels (or layers), level binding and level organization (linear, nested or partly nested).

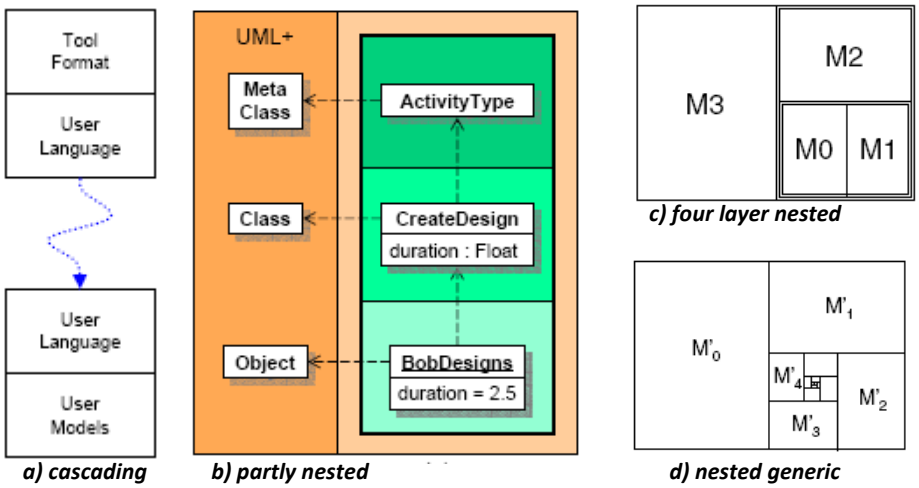


Figure 2-18 - Different modeling architecture designs (taken from [11] and [1])

The options that [11] presents are summarized in Figure 2-18. The binding of layers in a cascading architecture is loose as is shown in Figure 2-18a. Tools as Software Factories [38] use this approach where the modeling language is generated from the metalanguage specification. We merely include it here for completeness; because it is more an implementation approach then a conceptual approach for the modeling architecture. Figure 2-18b presents a partly nested architecture where the metalanguage is used as physical

definition for constructs at each layer. Figure 2-18b and c show the nested architecture with limited layers and with layer recursion ad infinum as MML has. Figure 2-14 completes the set of options with a strictly layered architecture.

2.5 Ontology and Modeling

The fields of Ontology and Modeling are related although they are not the same. Modeling is often done for pragmatic purposes. We make a model, when we draw an explanation on a whiteboard. This model does not have to conform to any precise language and can freely be interpreted.

Ontology, however, does not provide that freedom, since its goal is to represent the world as precise as possible as we saw in Section 2.2. It becomes obvious, that when combining the fields of modeling and metamodeling we have to keep this in mind. The formal constraints of Ontology cannot always be applied in modeling, because the languages do not always live up to the challenge of being ontologically correct. A simple example is the fact that UML and Java can define classes without any attributes. Another one is law 6 in [87] that states that properties should have values because "not having a property is not a property". Apparently, UML and Java cannot obey to this for practical purposes.

2.6 Conclusions

In the current chapter, we have treated Ontology, languages and MDE.

Ontology (capital O) was introduced here as a philosophy. From the field we took two important ontologies that are being used in computer science. From both of them we drew important concepts, which can be used to support concepts in the field of modeling.

Linguistics provides knowledge for computational languages. The distinction between *syntax* and *semantics* is present and important in both fields. Languages also form the pragmatics for modeling. Diagrammatic and textual languages are used for model and language input and output. Languages are related to Ontology in the sense that they make an *ontological commitment* to the world. Furthermore is language design a daunting task to balance expressiveness and precision.

In MDE modeling languages are a special kind of languages that describe both the model of the abstract syntax as well as the model for the semantic domain. In some modeling architectures these two instanceOf relations (linguistic and ontological) are made explicit. Because of these two instanceOf relations, modeling is full of relativity.

To support our use of the concepts we had to assume definitions for several modeling terminologies. The choice for the domain of Ontology allows us to make more concrete and detailed commitments to the meaning of the terms. From Chapter 4 *"An Ontology Grounded Language" on the* terms will be used and applied in our metamodeling approach.

Chapter 3 – Identification of Problems in Contemporary Modeling Architectures

3.1 Introduction

In the current chapter, we present the motivation behind our intention to propose a new metalanguage and thus a new modeling architecture. Contemporary metamodeling architectures already aim at providing a solution for modeling, metamodeling and model exchange. The purpose of these architectures is to provide a sound basis for modeling and metamodeling. In the current chapter, we also show that they cannot offer this in all respects.

3.2 Construct Incompleteness, Overload and Excessiveness

It is required for any language to define a mapping from the constructs to the real world [30]. Otherwise, a language suffers a ***lack of real-world relation***.

Traditional metalanguages like for example UML [75] and MOF [71] fail to do so. They implicitly take an arbitrary commitment to the object-oriented domain. This commitment may an appropriate one when using UML to model a software system. However, UML is a general-purpose language and thus needs to be suitable to model other domains as well. The same can be said for MOF. It is used to express modeling languages. However, does it have an adequate set of constructs to express languages? Is the meaning of the constructs unambiguous and can they be used in a consistent manner?

While this whole thesis aims at giving an answer to the first question, here we focus on the second question. We can answer the question for UML and MOF at the same time, because both use the same set of constructs. This is a consequence of their entangled history as described in the previous chapter. Thus, if we show a property of the UML constructs of `Class`, `Object`, `Attribute`, `Association`, etc, it holds for MOF.

The Appropriateness of Constructs in UML (and MOF)

Recent studies on UML [30][42] showed several inadequacies of this language regarding its modeling foundation. Both use a reference Ontology to analyze UML constructs for their appropriateness: *"a well-grounded, axiomatized upper level ontology is an important step towards the definition of real-world semantics for conceptual modeling diagrammatic languages"* [42].

Evermann and Wand [30] use BWW Ontology to analyze the language. Guizzardi et al. use a FCO ontology developed on their own effort. Both ontologies have been successfully applied before in information system technologies. Figure 3-1 shows that the preciseness of a reference ontology can capture the state of affairs that are admissible in a domain more closely than a modeling languages L1 and L2 could ever do.

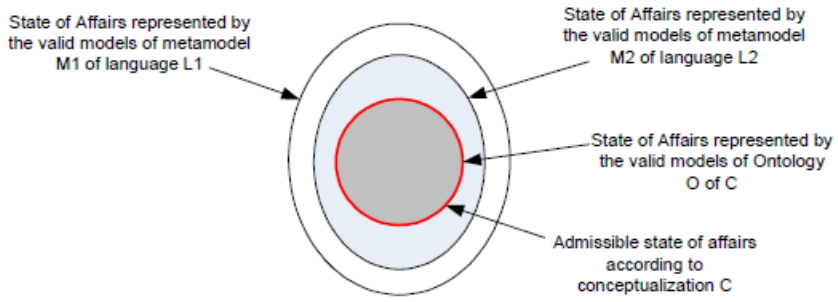


Figure 3-1 - A reference ontology to measure domain appropriateness(taken from [42])

The findings on the domain appropriateness of UML include:

Construct incompleteness – A first conclusion that is drawn from the comparison is the absence of constructs in UML to capture the *kind*, *role*, *quality* and *relator* that are in the reference ontology [42].

Construct inappropriateness – `Associations` cannot capture the real ontological nature of the relation concept. From the point of view of BWW, this is *construct_excessiveness* in presence of the `Attribute` construct, because the BWW ontology is rather minimalistic. Guizzardi's ontology is more extensive and therefore denotes this *construct overload*. We captured both under *inappropriateness*, whether they represent in the end exactly the same thing is we leave for the reader to decide.

Construct overload – Another ontological misuse of `Attribute` and `Association` can happen when a user decides to model, for example, a skill as an object. From an ontological point of view, this is a moment (or property in BWW). It breaks one of the laws that BWW enacted: "*Only Things can be modeled as objects*". Moreover, it results in inconsistent use of `Attribute` and `Association` as already mentioned in the previous point.

These represent just an overview of the findings that were done using ontologies. The results of the quoted papers go further. We choose the illustrative ones to demonstrate the validness of conclusions that can be drawn. When applying ontology to more complex and formal concepts like *aggregation* the results will be less obvious to interpreted, yet equally usable. Several other studies that use different ontologies and/or focus on different parts of the UML definition are found here: [45][43][26][67]. A conclusion is easy to establish. These deficiencies have the potency to result in *inconsistent* and *ambiguous* models and metamodels.

Poor Semantics Definition

UML and MOF are modeling languages that come with convenient graphical syntaxes. The whole purpose of their entanglement is partly to reuse this syntax. Especially in UML (and the related MOF), the same syntax is often reused to specify the semantics of the language itself. Some researchers have indicated that is this could result in weak separation between abstract syntax and semantic domain [46]. Since syntax and semantics seem closely related in MDA, this is cited as reason why both are not well described.

Furthermore, MOF makes the choice of using its abstract syntax to define itself. This seems to be a rather pragmatic choice (it allows the semantics description to reuse the same diagrammatic syntax) and to our knowledge, there is no verification of MOF against itself.

3.3 Multilevel Metamodeling

"*MOF and UML emphasize the linguistic dimension*", say Atkinson and Kühne [9]. For initial modeling architectures, this was fine; it allowed the enactment of an easy to understand four layered modeling architecture called MOF. However, the demands on UML and other modeling languages grew and soon MOF's foundations started to crumble. We describe in the current section some causes and effects of this process. First, we introduce some terms that are often used in the discussion about modeling architectures [9][7][5][10].

Terminology for Properties of a Modeling Architecture

Because meta and modeling languages are considered to be only structural definitions, they limit their semantics to the model at the layer below (M2 for metalanguages and M1 for modeling languages). Atkinson and Kühne call this *shallow instantiation*. The term *strict metamodeling* is used for the modeling architectures, which only allow shallow instantiation between the different layers and have a *linear* layer organization.

In the MOF architecture, each layer is defined in a layer above. The top layer is defined in itself. All layers thus have a lower and a higher layer. Through this, classes seem to play a double role in the modeling architectures. From the point of view of the layer above, they are objects. From the point of view of the layer below, they seem classes. This *Class-Object* duality can be observed in traditional modeling architectures. Atkinson and Kühne [9] explain how the syntax of modeling languages hides this dual nature of classes and objects. They introduce the term *Clabject* for it. The name of a class is displayed in its syntax, yet in reality, it is an instance of the attribute "name" from the defining class. To express this they created a cube to represent model elements (see Figure 3-2).

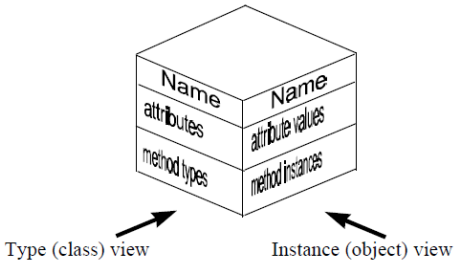


Figure 3-2 - A Clabject[8]

Ambiguous Classification

Multiple classification was an early problem in the interpretation of UML diagrams. As we explained in the previous chapter, each model element has multiple instanceOf relations. Because the linear hierarchy of the initial MOF architecture, these multiple relations had to violate the strict layered interpretation as is shown in Figure 3-3. The revised architecture

[8] From the combination of the word CLAss and oBJECT that are used in the UML jargon

that put user data models on the M1 layer partly solved this problem. It however deemphasizes the ontological instanceOf relation in which the user is interested [7].

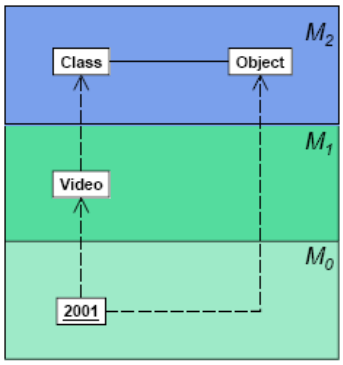


Figure 3-3 - Multiple classification in the MOF architecture (taken from [7])

Decreased Extensibility

There is a need for reuse of metamodels [27]. The semantics of UML assumes that users will only work at modeling levels M0 and M1. However, in order to reuse metamodels their users generally need to also model at level M2. UML supports this by means of *stereotypes*, which can be bundled into packages of stereotypes using *profiles*. A profile can be introduced to reuse the UML language for a specific architecture. Java only supports single generalization, whereas UML by default uses multiple generalization. By creating a profile Java that contains a special stereotype `<<JavaClass>>`[9] for `Class` the concept Class can be restrained to only one generalization.

Stereotypes are however, limited to annotation of types and the addition of static attributes called `Tags`. This ensures that the stereotype mechanism does not break the strict modeling hierarchy and that stereotyped models remain compatible with their originals [6].

Stereotypes can thus be seen as a restricted way to generalize the instantiation concept over all the levels. It becomes an alternative way to express instantiation without adding modeling power [5]. This adds extra semantic and notational baggage. Furthermore, it creates confusion when there are no rules offered on when to use which mechanism and why, which is the case with the current UML version. Especially since the instantiation mechanism choice is not meta-level independent, defining such rules may be complicated.

Replication of Concepts

Previous versions of UML suffered from a replication of concepts because they had to model the structure of instances for each `Class` individually (`Node`, `Component`, etc) [10]. This problem seems solved with the UML 2.0 specification [75], which introduces a general `InstanceSpecification` construct. It is up to tool vendors to represent classes on this construct according to the UML semantics.

[9] A notational convention for stereotypes in UML is to write them between smaller than/greater than symbols

Failure to Express Power Types

"A power type is a type the instances of which are subtypes of another type (called the partitioned type)" [76]. A classical example is the `TreeSpecies`, which instances can be `Elm`, `Oak`, etc. Obviously the latter ones are all types themselves. The partitioned type in this case is the `Tree`. The benefits of power types are that new types can be introduced dynamically, as instances. At the same time, this makes it hard to support power types, since they are also *specializations* of the partitioned type. And - this is the problem - *specialization* is normally done statically in modeling. Figure 3-4 shows this in an example with `Vehicles`, `VehicleKind` and `Boat`.

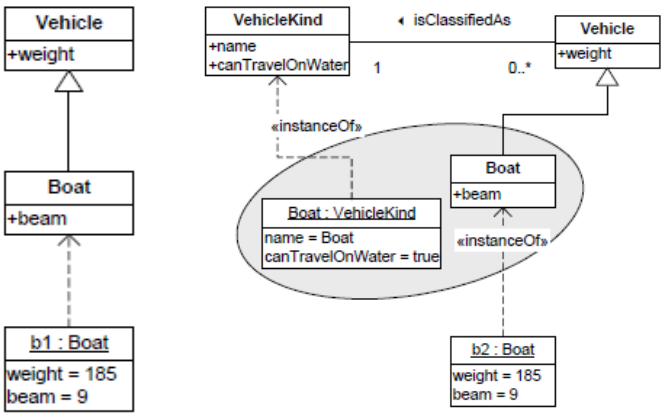


Figure 3-4 - An example power type (taken from [47])

Instantiation Semantics

Atkinson and Kühne tried to "compensate for shallow instantiation" in order to give semantics to MOF that keeps intact its *strict* modeling hierarchy. They feel this hierarchy is important, because *"the strictness discipline was instrumental in uncovering and understanding the subtle problems of the original presentation of the metamodeling framework. In the presented two dimensional framework, the strictness discipline is fully applicable and – like any guideline – provides help in staying away from unclear scenarios"*

To compensate for *shallow instantiation* they evaluated the use of *deep instantiation,* an instantiation mechanism over multiple layers in modeling hierarchy. *Potency* is introduced to express the number of levels that the instantiation may cross. *Dual fields* naturally appear under these circumstances, because fields can be both attributes and slots (holding values).

3.4 Language Independent Model Handling and Structure

In previous versions of MOF, the M0 layer was defined for user data models. MOF did not provide a *language independent structure* for M0, which made the interpretation of this layer implicitly dependent on the modeling language [57][65]. Now that the M0 layer is assumed to be the real world and the user data models are incorporated in the M1 layer, the structure is there. However, from the point of view of the metalanguage, there is no explicit notion of the relation between models and model elements in M1. For example, the UML language can tell us which object is an *instance of* which class. MOF, however, cannot provide us with this

information, since it is oblivious to the instantiation semantics of UML (or in the general case, any other modeling language).

The absence of a language independent structure for user data makes the handling of this layer in the modeling hierarchy dependent on the modeling language. Furthermore MOF provides an instantiation mechanism which is used for modeling languages expressed in it, but which is separate and may be different from the mechanism that the languages itself use. There is no precise definition about how these two instantiation mechanisms cooperate on the lowest modeling layer, the user data layer [62][21][37].

3.5 The Adverse Effects of the Problems on Automation in MDE

Model Transformations

On one hand the identified problems in (meta)modeling result in less interoperability than may be possible. Tools still needs to hardcode the instantiation mechanism from MOF and the metalanguage separately. On the other hand, it limits recurrent MDE tasks. These tasks include model transformation and model querying.

The OMG specification for QVT [71] supports only a limited set of model transformation scenarios. Firstly, heterogeneous data translation is not supported [61][62]. Heterogeneous refers to the fact that data can be stored with different technologies. In Figure 2-17, we showed a few example including XML and RDF. Figure 3-5 gives a concrete example of a transformation between a *database model* and an *object-oriented model*. The difficulty is to derive the transformation of *T2* from the transformation definition *T1*.

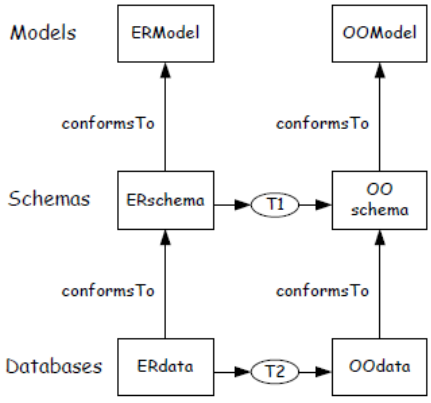


Figure 3-5 - The data translation problem in model transformations (taken from [61])

Model Querying

The OMG standard for model querying OCL [73] is limited to MOF and UML models [57]. If MOF would provide a real metamodeling environment, such a query language would have to be language independent. That model querying and navigation is dependent on the instantiation semantics of the modeling language is well known [36][57][37].

3.6 Analysis of the InstanceOf Relation

The described problems all relate to a lack of *Uniform treatment of language structures.* Especially the instanceOf relation seems to play an important role in all the described

problems. Therefore, we will investigate its role in the modeling architecture in the current section.

It has been pointed out several times now that instantiation is not only *linguistic*, but also *ontological*. The ontological instanceOf is only implicitly known via the semantics of the modeling language. This relation's dual role may cause ambiguity [18]. Yet in traditional modeling architectures, the instanceOf concept only plays a secondary role. To gain more knowledge about the nature of the instantiation in the modeling architecture, we look at the architecture from different perspectives in the next subsections.

From the Perspective of Metamodeling

We recognize intensional and extensional models. In UML, these are class diagrams and object diagrams, in OWL Schema's and RDF models, in database technologies table schemas and rows. However, from the metamodeling perspective, these are all just models. The metametamodel does not distinguish intensional and extensional models in its *ontological commitment*. Therefore, from the perspective of the metalanguage, we see the models and metamodels (see Figure 3-6). The semantics of the metalanguage provides an ontological instanceOf between the two.

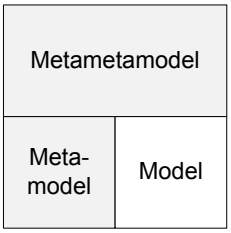


Figure 3-6 - The models from the perspective of the metalanguage

The distinction between intensional and extensional models will become clear when we look from the perspective of modeling.

From the Perspective of Modeling

The metamodel does see the intensional and extensional models. UML "sees" class diagrams and object diagrams, while OWL "sees" Schema's and RDF models. The *ontological commitment* of the metamodel is thus what is in the intensional model and what is in the extensional model. The semantics of the modeling language provides an ontological instanceOf between the two.

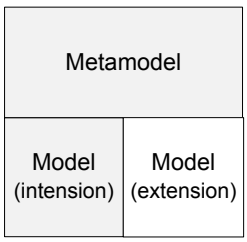


Figure 3-7 - The models from the perspective of the modeling language

From the Perspective of Modeling Architecture

The ontological instanceOf relation that exists by virtue of the semantics of the metalanguage is a linguistic instanceOf for the modeling language. The same *relativity* in modeling that was generally described in Chapter 2, contributes to ambiguity in the whole modeling architecture. Current interpretations of this relation do not do justice to its nature. Without a unified semantics for the instanceOf relation for the whole architecture, real metamodeling is simply not possible as we see from the described problems.

3.7 Conclusions

We can conclude that all problems are related to metamodeling and especially its instantiation semantics:

- A lack of ontological grounding of metamodel constructs results in expressiveness problems with these constructs,
- Because of a failure to find a good interpretation for the metalayers as well as the instanceOf relation, multilevel metamodeling causes problems,
- The instanceOf relation has to be hardcoded for metalanguages and modeling languages and there is a lack of structural definition for model elements,
- Automation techniques in MDE cannot be generalized because of their dependency on the (fixed) instanceOf semantics of the metalanguage,
- The instanceOf relation is not well described by the metalanguage and consequently by the modeling languages.

Chapter 4 – An Ontology- Based Modeling Architecture

In the current chapter, we present an ontology-based metalanguage, which is capable of expressing languages and their models together with their generalization and instantiation semantics. The characterization of this language is that can recognizes multiple instanceOf relations between models and express their instantiation semantics. This provides a basis for unambiguous multilevel modeling and shows how linguistic and ontological instanceOf relations are related.

We keep the view that languages can be represented by their abstract syntax and add the capability to specify an instantiation mechanism for these structures. This overcomes the major drawback of modeling architectures that the instantiation mechanism is hardcoded into the tools. In effect, (modeling) languages can become more tool independent.

4.1 Introduction

Chapter 2 gave an overview of the concepts in modeling and ontology. The goal of a modeling language was to capture a part of the world. Of reality. Therefore, most languages focus on a specific part of this world. UML on computer science, BPEL on business processes, etc. MOF tries to increase tool independence by providing a general structure on which to express both languages and models. However, in Chapter 3 we saw how these modeling architectures are ill defined and unsuitable for multilevel modeling. This motivates us to create a new metalanguage and thereby a new modeling architecture.

In the current chapter, we present the basic ideas behind the *Ontology Grounded MetaLanguage* (OGML) . Section 4.2 explains the *approach* and design decisions for the language. Concepts and reasoning are drawn for Ontology here because this is we identified it as a knowledge domain for metamodeling.

The subsequent section 4.3**,** presents *the metalanguage OGML*. The approach chosen here is to explain *modeling* and *metamodeling* practices at the same time. This is done with a small example, which covers most of the modeling architecture. In between the examples, the abstract and concrete syntax of OGML is explained.

In section 4.4, we propose a general *structure for modeling space,* which can represent all models in the architecture. This structure is based on ontological classifications and we call it *OGML eXtensional* (OGMLX). This structure can provide is the uniform base to represent all models on; for both models and languages.

As a metalanguage for modeling languages, OGML has the capability to describe itself. Certain interesting aspects of this *conceptual self-reflection* are given in section 4.6. Subsequently section 4.7 builds on these insights to *prove* that the modeling architecture indeed represents every model on the OGMLX structure. Section 4.8 draws *conclusions* from the proposed architecture and the proof we presented.

4.2 Approach

Kurtev proposed the conceptual foundation behind OGML [63]. Like him, we adopt the solution domain of Ontology (see also the approach in Chapter 1). As it is the study about the possible world structures, Ontology is our primary knowledge source for deriving the primitives of the metalanguage. This choice can only directly solve the constructs expressiveness problems that we described in the previous chapter. To solve the problems caused by the *instantiation semantics* we have to take further steps. We will describe this in the current section.

The Primitives of the Metalanguage

The purpose of the metalanguage is to support modeling languages. For example, UML "sees" the phenomenon in the real world as Objects, which are instances of exactly one class. OWL "sees" the world as resources, which can be instances of multiple classifications. Similar observations can be made when we metamodel data representation languages like RDF and XML [24]. The existence of universals is thus presumed, *a priori*, in the ontological commitment of all these languages. Therefore, *we propose the use of FCO for our modeling primitives*. The insights on the ontological meaning of properties that stem from using the *BWW* ontology will not become less valuable using this approach as we already mentioned in Subsection 0. This way we solve the problem of ***lack of real-world relation***.

The Semantics of the Metalanguage

It became apparent in the previous chapter that several problems are caused by the under-specification of the instanceOf relation. Therefore, we propose to *lift instanceOf semantics to a first class concept* in the metalanguage. This raises the level of abstraction of the language from structural definition to semantics definition and solves the problem described in Chapter 1 as: ***lack of language constructs*** *and* ***lack of modeling constructs***.

This way we can realize *language dependent instantiation semantics*, which are captured in a uniform way. The real nature of the languages becomes more evident: a metalanguage defines instantiation between modeling language and models and the modeling language defines instantiation of intensional and extensional models.

Because the metalanguage itself is also a modeling language, we can express it in itself. In Chapter 2, we mentioned the limitations of this approach. Both structurally and semantically there need to be grounding on existing solutions in order to avoid inconsistencies.

Model Structure

The adoption of *FCO* has one important consequence: InstanceOf relations have to be recorded on model level. Contrary to BWW, where classes (*Kinds)* play a secondary role and are established via set inclusion of properties, in an *FCO* ontology the class is determined *a priori* and thus fixed for each construct. The modeling space [62] this needs to provide a property to record this relation.

Defining a Precise Scope

The basic ontology presented in section 2.2.4 is simple and does not accommodate significant part of the available ontological knowledge. The ontology does not consider three

fundamental concepts: time, space and part-whole relation. For the latter one, there exists well-developed theory called *mereology*. It is difficult to decide which concepts to be taken into account when an engineering solution needs to be crafted. We opt for these four basic categories and the relations among them as the first step in our experiment in applying ontological categories in defining metamodels. Missing concepts should be defined per metamodel if needed.

On the Use of Ontology

In section 3.2, we gave an overview of some ontological inconsistencies of modeling languages. Some constructs cannot capture the expressiveness of their real-world equivalents and some relations do not obey to logical laws established in Ontology as discussed in Section 2.5. We still intent to be able to express all modeling languages without significant modifications, which would change their semantics. We choose, therefore, a pragmatic approach in the use of Ontology. In the general case, we will use ontological reasoning for the OGML semantics. Where modeling practices require a deviation from it, we introduce our own semantics and explain the deviation.

4.3 The Metalanguage OGML

In the current section, we introduce OGML by applying it in an example. The concrete syntax of the language can be found in Appendix B. Before each step of the example, we will explain a part of the semantics. We focus here on the description on the abstract syntax and give semantics in natural language for each construct. Later, in Chapter 6 a more formal description of the semantics is given.

For the purpose of the example, the UML language is used. We only focus on a subset of UML, which is most relevant for our day-to-day modeling operations. We could call this language therefore "*SimpleUML*". The metamodel of the language is shown in Figure 4-1 (left part) together with an example model (right part of the figure) and instanceOf relations. The upper part represents class diagrams and the lower part object diagrams.

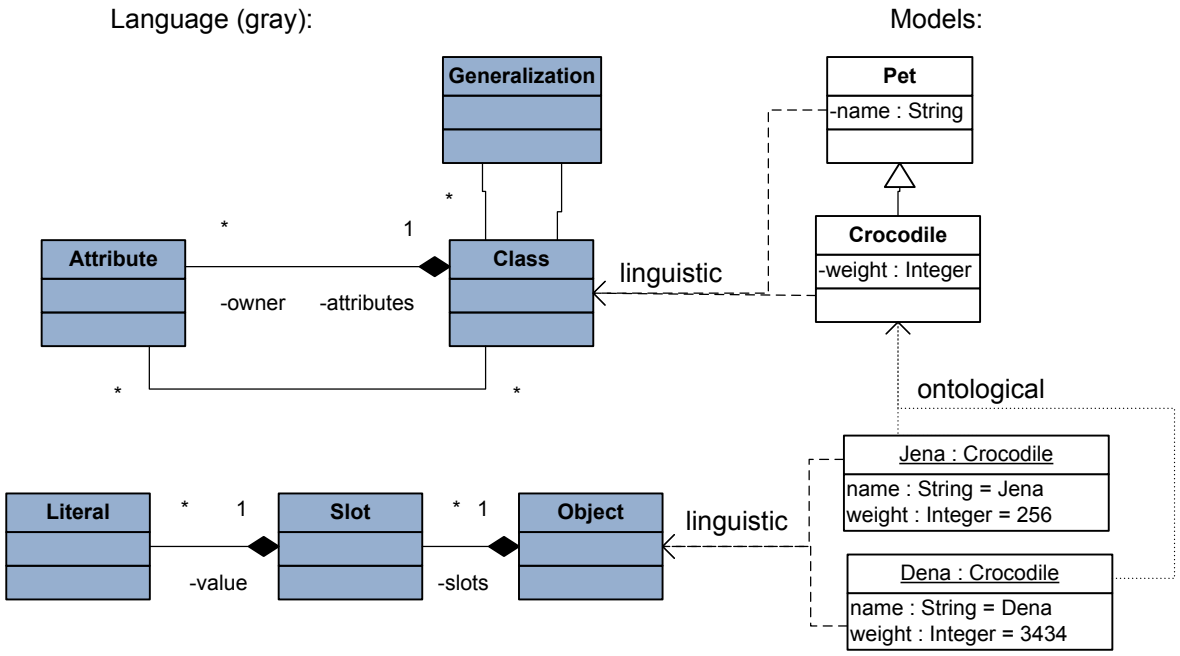


Figure 4-1 - The example language SimpleUML

Thus, the exercise will be to express the diagramming capabilities of (Simple)UML in OGML. In the following subsections, we will use the pattern of first introducing the OGML constructs and then applying them in practice repeatedly until all constructs are explained.

4.3.1 Language Constructs

OGML provides a set of language constructs or Definitions. These are based on the categories of *FCO*. Figure 4-2 shows these constructs. It is followed by a short introduction to their semantics, which is based on the description of FCO in Section 2.2.

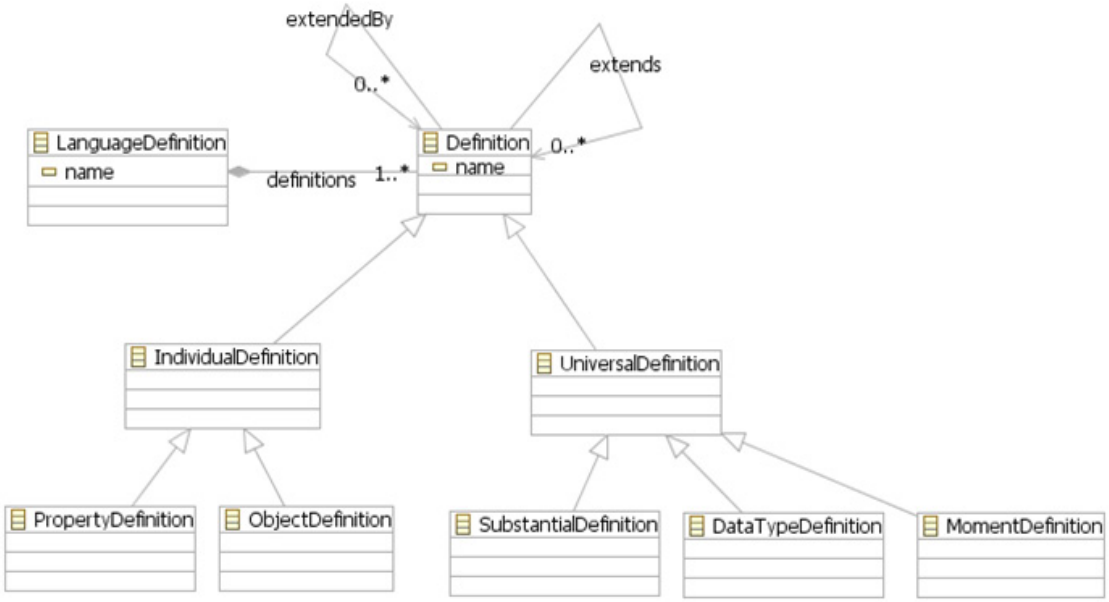


Figure 4-2 - OGML's language constructs

LanguageDefinition

A LanguageDefinition (LD) is a group of language constructs, relational constructs and semantic definitions (which will be introduced in following subsections). This construct is not based on a category in Ontology but is needed for the pragmatic purpose of bundling language definitions. A language definition has a ***name***.

Definition

A definition is the generalization of all language constructs. It is *abstract* and thus not used as language definition. A Definition has:

name, a unique name within the language

extends, a set of language definitions that are specializations

extendedBy, a set of language definitions that extend this definition

UniversalDefinition

A UniversalDefinition (UD) is a generalization of the definitions that are used as *universals* in a model. It is *abstract* and thus not used as language definition. It is a specialization of *Definition*.

IndividualDefinition

An IndividualDefinition (UD) is a generalization of the definitions that are used as *individuals* in a model. It is *abstract* and thus not used as language definition. It is a specialization of *Definition*.

SubstantialDefinition

A SubstantialDefinition (SD) is used to define language constructs for the *substantials* in a model. If a model contains substantials, it can be used as an intension for other models.

A SubstantialDefinition is a specialization of *UniversalDefinition*.

MomentDefinition

A MomentDefinition (MD) is used to define language constructs for the *moment universals* in a model. Since moment universals characterize universals it is related to a UD.

A MomentDefinition is a specialization of *UniversalDefinition*.

ObjectDefinition

An ObjectDefinition (OD) is used to define language constructs for the *substantial individuals* in a model. A model with only individuals is an extensional model.

An ObjectDefinition is a specialization of *IndividualDefinition*.

PropertyDefinition

A PropertyDefinition (PD) is used to define language constructs for the *moments* in a model. *Moments* represent relations between individuals in a model.

A PropertyDefinition is a specialization of *IndividualDefinition*.

DataTypeDefinition

A DataTypeDefinition (DTD) is used to define *data types*. A DTD can be used as type for a *substantial property that is instantiated to an intrinsic property* (see subsection "Relational Constructs"). In terms of modeling languages, we usually refer to this as *the type of a literal*.

Abbreviations for the Language Constructs

Table 4-1 summarizes the abbreviations we used for OGML language constructs.

Table 4-1 - Abbreviations for language constructs of OGML

Full Name	Abbreviation
LanguageDefinition	LD
UniversalDefinition	UD
IndividualDefinition	ID
SubstantialDefinition	SD
MomentDefinition	MD
ObjectDefinition	OD
PropertyDefinition	PD
DataTypeDefinition	DTD

Instantiation of Language Constructs

Here we give an overview of the different instantiation of the languages constructs that were just introduced. We do this in terms of a partly nested modeling architecture, but the reader should keep in mind that from this picture no conclusions can be drawn about the OGML modeling architecture. Figure 4-3 gives schematic representation for the instantiation semantics. Instantiation is represented by arrows. At each "layer", an instantiation is represented by a small box with a number to indicate the depth of instantiation of the construct. *It can be seen in the figure that we treat linguistic instantiation exactly the same as the ontological instantiation* between the intension and extension.

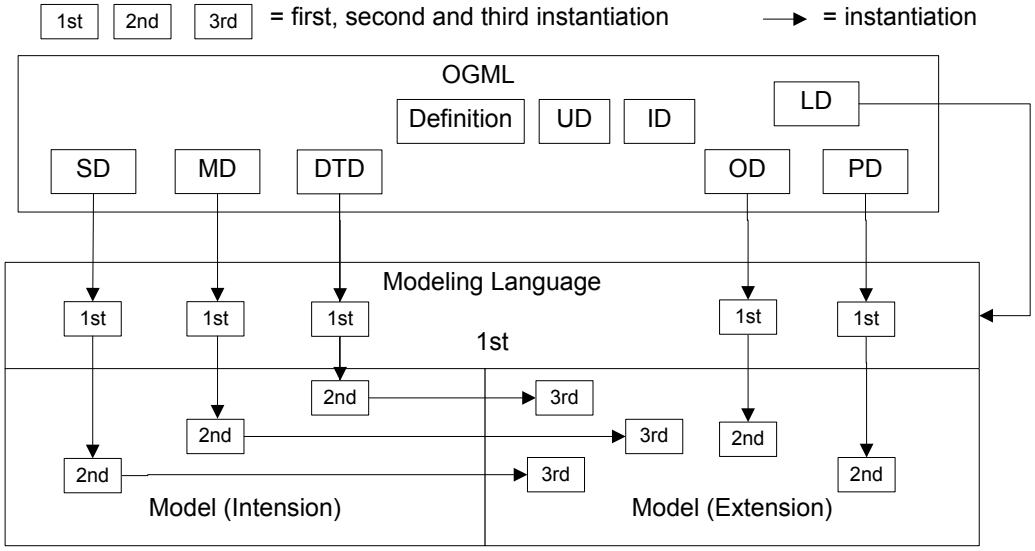


Figure 4-3 - A schematic view of instantiation semantics for OGML language constructs

Defining the SimpleUML Modeling Language

To define SimpleUML we define a Language with the name "SimpleUML". This is done in Listing 4-1.

```
Language SimpleUML {
        ...
}
```

Listing 4-1 - OGML by example: defining the language SimpleUML

Defining the Universals of SimpleUML

Now we can define constructs for classes and attributes. A class, as for example the Crocodile `Class`, is a *substantial* from *ontological point of view*, because it is a prototype for all possible crocodiles ("Jena", "Zena", etc). Therefore, we declare a class as SD (see Listing 4-2). Classes have attributes, which are in turn *moment universals*, thus declared as MD's in the language definition. To be able to declare literal types in the intensional model we add a DTD "UMLDataType". It may be used in the extensional model to instantiate literal value.

```
SubstantialDefinition Classifier {
        ...
}

SubstantialDefinition "Class" extends Classifier {
        ...
}

DataTypeDefinition UMLDataType extends Classifier {
        ...
}

MomentDefinition Attribute {
        ...
}
```

Listing 4-2 - OGML by example: defining the language constructs for universals

Defining the Individuals of SimpleUML

Instances of classes - e.g. specific crocodiles – don not "just dangle in the air", but need to be represented as modeling construct. For this purpose we define the constructs for an extensional SimpleUML model (Listing 4-3) representing the object diagram model. It contains objects and slots just as in the UML specification. The reasoning for the ontological types goes similarly to the reasoning we used for the definition of the universals of the language. The difference is that we use *ODs* and *PDs* here instead of *SDs* and *MDs*. Literals will be represented as *ODs*[10].

[10] Here we deviate from *Rule 1* in [90], which stipulates that only entities can be represented as substantial individuals (in FCO terminology). A literal is a data value, has no identity, and thus is not an entity according to the dictum of Quine that *"no entities without identity"* [48]. The current version of OGML does however not provide extensive support for literals. For simplicity, all data values are stored as String and literals are not typed. It will be future work to provide a full and correct interpretation of literals.

```
ObjectDefinition Object {
        ...
}

ObjectDefinition Literal {
        ...
}

PropertyDefinition Slot {
        ...
}
```

Listing 4-3 - OGML by example: defining the language constructs for individuals

4.3.2 Relational Constructs

The previous subsection showed how languages should define universals and individuals. To be able to define meaningful models a language should also provide means to connect them together. In this manner a graph-like structure can be formed which is so typical for models and data structures. *Attributes*, *CharacterizationRelations* and *InherenceRelations* provide the needed relational constructs. These constructs are based on the notion of *(substantial) properties* in Ontology (see Subsection 2.2.4). Figure 4-4 shows the abstract syntax of these constructs.

From the figure is becomes clear that all these relational constructs refer to multiple *Definition*s. These ranges are introduced because it is ontologically incorrect to use generalization between the different categories [39] (see subsection 4.3.4). However, it is still desirable to have attributes refer to, for example, a substantial and an individual. For example, RDF allows references from a model to the RDF Schema, which is also an RDF model [89]. Therefore, ranges are used instead of types. In addition, this approach will guaranty better support for modeling languages; languages that support multi-typed properties can be expressed in OGML.

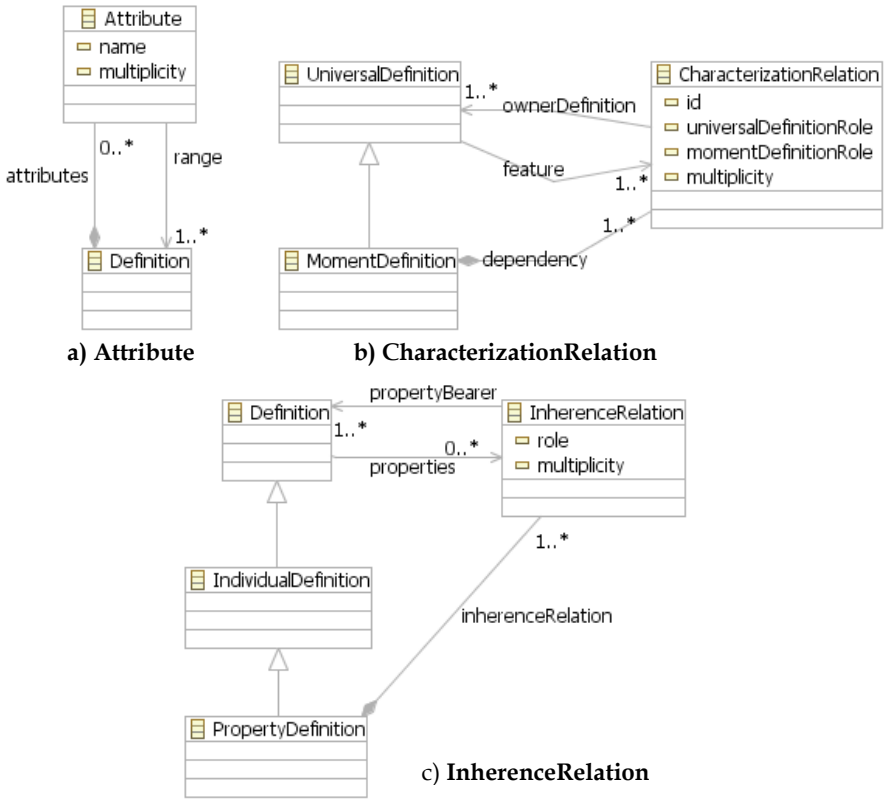


Figure 4-4 - OGML's relational constructs

Attribute

An Attribute (shown in Figure 4-4a) can represent *structural properties* of both *individuals* and *universals* in models.

An attribute has a ***name***, a ***multiplicity*** and a ***range***. The range defines the type (or set of types) of the attribute. Attributes also have an ***ownerDefinition*** which is the Definition where they attribute the property to (this is not shown in Figure 4-4a, but is the opposite of *attributes*).

CharacterizationRelation

A CharacterizationRelation (CR) should represent *mutual properties* for *universals* in models, which are part of the ontological perspective that the language makes. Expressed in the language definition, it thus relates MDs to SDs is can be seen from the abstract syntax shown in Figure 4-4b.

A CR has an ***id*** and a ***multiplicity***. Since it connects two constructs, a CR has two roles: the ***universalDefinitionRole*** (in the direction of the SD) and a ***momentDefinitionRole*** (in the direction of the MD). A CR is thus a bidirectional relation.

InherenceRelation

An InherenceRelation (IR) can represent *mutual properties* for *individuals* in models, which are part of the ontological commitment that the language makes. This is done unidirectional.

Expressed in the language definition, it thus relates ODs to PDs is can be seen from the abstract syntax shown in Figure 4-4c.

An IR has a ***role*** for the PD and a ***multiplicity***.

Abbreviations for the Relational Constructs

Table 4-2 summarizes the abbreviations we used for OGML language constructs.

Table 4-2 - Abbreviations for relational constructs of OGML

Full Name	Abbreviation
CharacterizationRelation	CR
InherenceRelation	IR

Instantiation of Relational Constructs

Here we give an overview of the different instantiation of the relational constructs that were just introduced. Figure 4-5 gives schematic representation for the instantiation semantics. Instantiation is represented by arrows. At each "layer", an instantiation is represented by a small box containing the depth of instantiation.

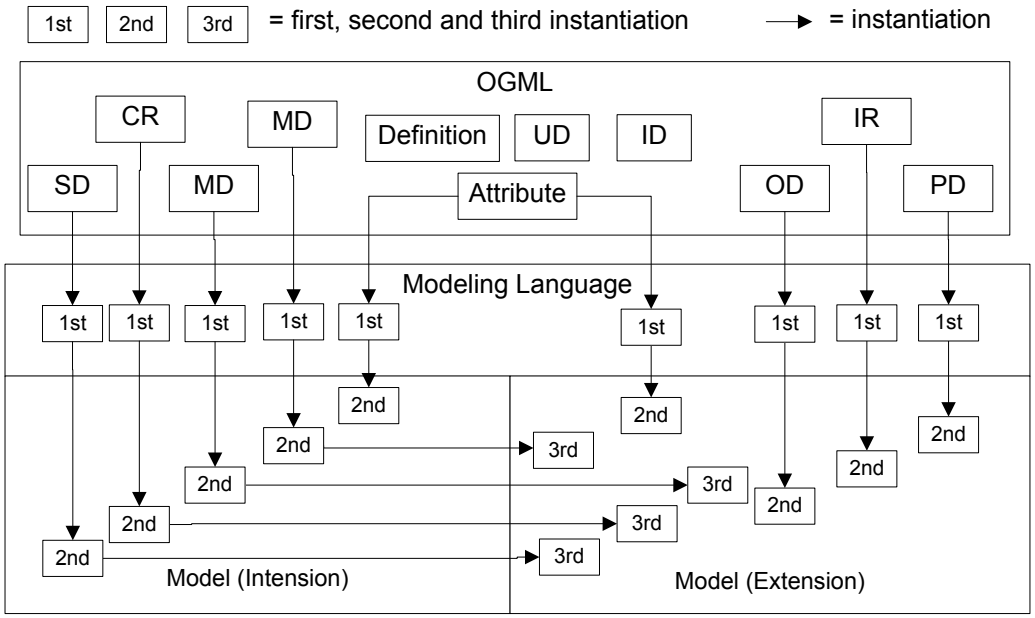


Figure 4-5 - A schematic view of instantiation semantics for OGML relational constructs

Defining Structural Properties for SimpleUML Constructs

`Classes` and `Objects` in UML have names and other structural properties. We introduce here the structural properties, which are *exactly those properties, which are not part of the ontological perspective that the language semantics provide.*

Structural properties do indeed not contribute to the ontological perspective directly. They are merely used indirectly to establish the ontological commitment. This can be easily explained with the example of UML from the point of view of a modeling tool. When querying an UML model, the tool only handles *objects* and *slots*. An *object* can have a whole

array of *slots*, but the query is only interested in the slots that belong to one UML `Attribute`, e.g. the attribute "weight" of Crocodile (queries are specified against the class model). The tool now has to traverse all *slots* in order to find the one with property "name" equal to "weight". Other examples of the relation between structural and ontological model handling are provided in [57] and [37].

From the point of view of Ontology, we are however just also talking about *structural properties* for the language we are defining. Listing 4-4 shows how they are added as `Attributes` to the language constructs according to the UML metamodel. These attributes are not to be confused with the `Attribute` as it also appears in the SimpleUML metamodel.

```
SubstantialDefinition Classifier {
        attribute name : "String";
}

SubstantialDefinition Class extends Classifier {
        attribute isAbstract : "Boolean";
}

DataTypeDefinition UMLDataType extends Classifier {}

MomentDefinition Attribute {
        attribute name : "String";
        attribute lowerbound : "String";
        attribute upperbound : "String";
        attribute type : Classifier;
}

ObjectDefinition Object {}

ObjectDefinition Literal {
        attribute value : "String";
}

PropertyDefinition Slot {
        attribute name : "String";
        attribute value : Object, Literal;
}
```

Listing 4-4 - OGML by example: defining the attributes for universals and individuals

Defining Ontological Properties for SimpleUML Constructs

Here we deal with the moments and moment universals and how they are related to individuals and universals. These constructs will provide a basis for the *ontological perspective* that we are realizing with SimpleUML.

From the point of view of Ontology, the moment universals are *substantial properties*. All moment universals characterize in at least one substance [87], so a MD needs to "characterize" a SD. The CR is defined for UML Attribute in Listing 4-5. It defines a named bidirectional relation between `Class` and `Attribute` with a multiplicity to ensure that a class has at least one attribute. Later we will introduce the `Attribute` and CR, for now it is sufficient to know that it relates universals in models. We continue with a structure on which instantiated classes can be expressed.

```
MomentDefinition Attribute {
        ...
        attribution universalDefinition = "Class"
                universalDefinitionRole = "owner"
                momentDefinitionRole = "attributes"
                multiplicity = 1-*;
}
```

Listing 4-5 - OGML by example: defining the characterizations for universals

From the point of view of Ontology, the moments are *(individual) properties*. An IR connects moments to individuals. As shown in Listing 4-6 (the *dependsOn* line represents the IR).

```
PropertyDefinition Slot {
        ...
        dependsOn Object, Link role = "slots" multiplicity = *;
}
```

Listing 4-6 - OGML by example: defining the inherence for individuals

What is in the Model?

After introducing all the new language constructs for SimpleUML, it is a good moment to take a look at what may be in the models. Figure 4-6 is a concretezation of the previous figures that showed SimpleUML. The model layers are filled with the constructs from the example model with a class `Crocodile` and two crocodiles and only their weights. Attributes ad intrinsic properties are only represented with arrows that point to strings and mutual properties are represented with thick lines between the model constructs.

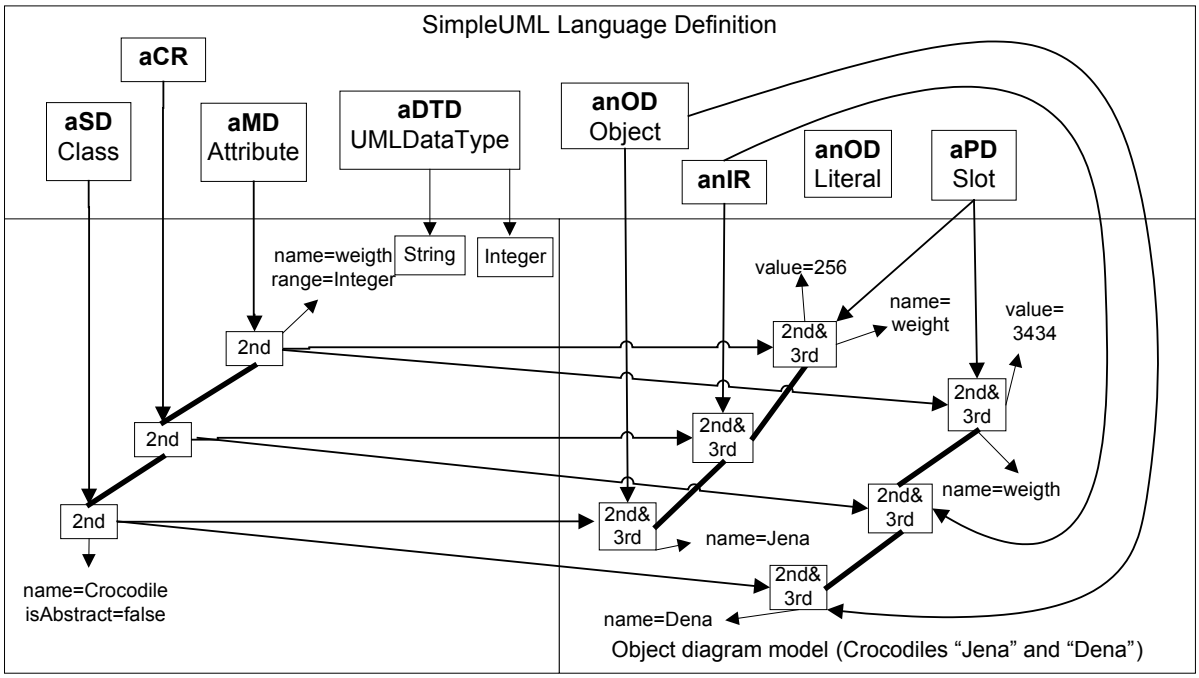


Figure 4-6 - A schematic view of SimpleUML with the example models

A difference with the previous pictures is the combination of constructs 2 and 3 into one construct. Later it will become clear what defines this equivalence. The instanceOf relations

of the Integer, String and Literal are not shown in the figure because not all their relations will fit in. Basically every concrete value ("weight", "Jena", "Dena", 256 and 3434.) in the Object Diagram model is an instance of both Literal (linguistically) and Integer or String (Ontological). The intrinsic properties of the Class Diagram model are also have the two instanceOf relations, how this works will become apparent in the end of the current chapter.

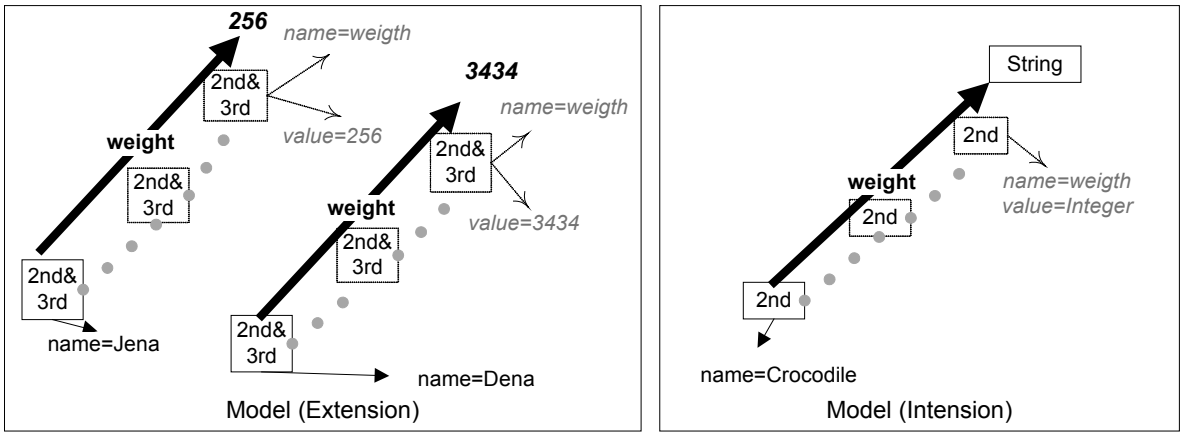
4.3.3 Ontological Perspective Constructs

SimpleUML is a language. Its Ontological Commitment consists of `Classes`, `Attributes`, `Objects` and `Slots` as we have seen. (Linguistic) instances of these constructs are in the models. Because SimpleUML is a *modeling* language, it makes an *Ontological Commitment on the level of the models*. From the point of view of Ontology, this means that a *different perspective* is taken on one model (extension) where another is used as ontology (intension). Instead of seeing `Object`, we want to see `Crocodiles` in the extension. Instead of seeing the *structural properties*, we want to see the *ontological properties*. To refer back to the statement from Webber et al.: "The properties exist whether humans perceive them or not" [87], likewise both kinds of properties exist in the extensional model. The ontological perspective will show ontological properties in the model.

In our OGML definition of SimpleUML, we can change the view on properties in a *consistent* and *controlled* manner because *we have defined ontological properties with different constructs than the structural ones*. *Structural properties* were expressed using (OGML) *attributes* and are moments in the model as shown by the arrows to strings in Figure 4-6. *Ontological substantial properties* were expressed using MDs and are related to SDs via CRs. *Ontological individual properties* were expressed using PDs and are related to ODs via IRs. As shown by boxes linked with thick lines in Figure 4-6. Figure 2-2 is also relavent here, because it shows these characterization and inherence relations on the universal and individual levels.

Thus, to change our perspective on an extensional model to the ontological one that is defined by SimpleUML, we only have to recognize another set of constructs as the properties. This is similar to working of 3d glasses which filter colors for the left and the right eye in order for our brain to combine the three dimensional information that is encoded in the motion picture. Here we have "to put on glasses" that filter out some constructs and highlight a combination of others.

The constructs that need to be highlighted are the moments that are connected via IRs as our new properties. The ones that need to be hidden in the ontological perspective are the structural properties. This is illustrated in Figure 4-7 where the new "virtual" moment is drawn over its structural definition. Exactly the same thing can be done in the intensional model with the moment universals and the CRs there. Analogous to Figure 4-7 we will find there that the substantial universal `Crocodile` has a substantial property `weight`, which is represented as an `Integer`. In UML terms, Class `Crocodile` has an attribute `weight` of type `Integer`.



a) **extensional model** b) **intensional model**

Figure 4-7 - The ontological perspective that UML provides on models

However, as Figure 4-6 already showed, some constructs need to be in place:

Need 1 - the instances of universals (e.g. the class `Crocodile` and its UML attributes) need to be *mapped on* the same constructs as the *instances of the IDs* (e.g. objects and slots),

Need 2 - the *instanceOf* must be *established* between the constructs of extensional model and intensional model and between the models and the language,

In addition, tools that support the pragmatics of model input, need to know how intensional models are instantiated. In a linear architecture, this is easy to implement with a parser as shown in Subsection 2.3.1. In the presence of a second instanceOf relation, its constraints on the moments in model need to be expressed. This information can however also be used for model querying, just like the *navigation semantics* is expressed in the UML specification. Therefore, we add:

Need 3 - the value needs to be identified and type and multiplicity need to be checked to support *navigation*, *conformance checking* and *instantiation* of models.

OGML supports the definition of these semantics by means of the *InstanceOfDefinition*. It will be explained here. In the abstract syntax definition in Figure 4-8, the related constructs are shown. An *InstanceOfDefinition* relates a UD to a Definition and can contain several *AttributeFuntions* and *CharacterizationInstantiations*.

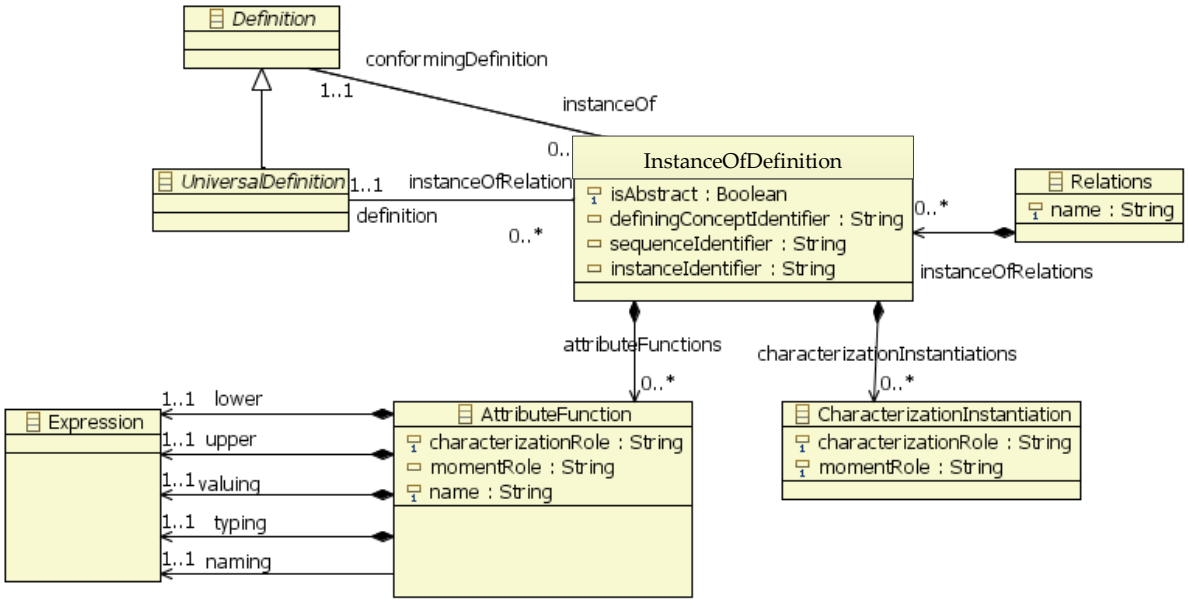


Figure 4-8 - OGML's ontological perspective constructs

Relations

Relations is merely a container to bundle the InstanceOfDefinition in the language. *Relations* can have a ***name*** and ***instanceOfDefinitions*** pointing to the set of InstanceOfDefinitions in the language definition.

InstanceOfDefinition

An *InstanceOfDefinition* (IOD) represents the semantics of the *instanceOf* relation *between* the *universals* and the *individuals* in models. Universals are called *instances* here. Individuals are called *defining concepts* here. This construct directly supports requirements Need 1 and Need 2 by providing the mapping between defining concepts and instances.

We can distinguish a *moment IOD*, which relates moment universals to moments, and a *substantial IOD*, which relates substantial universals and substantial individuals

Among the properties of the IOD are several *identifiers*, these can be used by expressions as variables. The IOD has the properties:

definition, the type of the defining concept,

conformingDefinition, the type of the defining concept

isAbstract, "true" if the IOD is abstract, in which case it is not directly used for instantiation, but indirectly via the *GeneralizationRelation* (explained in the next subsection). Generalization allows "inheritance" of *AttributeFunctions* and *CharacterizationInstantiations*.

definingConceptIdentifier, a variable name for the *defining concept*. This identifier can be bound to a defining concept in the intensional model, which conforms to ***definition***.

sequenceIdentifier, a variable name for multiple *instances*. This identifier can be bound to a set of instances in the intensional model, whose elements conform to ***conformingDefinition***.

instanceIdentifier, a variable name for the *instance concept*. This identifier can be bound to a instance in the intensional model, which conforms to ***conformingDefinition***.

attributeFunctions, points to the set of *AttributeFunctions*

CharacterizationInstantiations, points to the set of *CharacterizationInstantiations*

Instantiation may be of different kinds. This is different for each language; OWL [90] allows an instance to be instanceOf multiple defining elements for example, while UML allows only one. To support multiple instantiation we can introduce a ***universalMultiplicity*** and ***individualMultiplicity*** here. However, this currently remains *future work*.

Constraints:

- Only *moment IODs* and *substantial IODs* exist. A moment IOD can only contain a MD as definition and a PD as conformingDefinition. A substantial IOD can only contain a SD as definition and a OD as conformingDefinition[11].

CharacterizationInstantiation

A *CharacterizationInstantiation* (CI) represents the *instanceOf* relation *between IRs* and the *CRs* in models. Since these are connected to respectively *individuals* and *universals*, it is logical that a CI is contained by an *IOD*.

It has several attributes:

instanceOfRelation, points to the containing IOD

momentRole, is a by-name reference to an IR (role).

characterizationRole, is a by-name reference to a role in a CR. A CI together with AttributeFunctions realizes Need 3**.**

Constraints:

- The IOD, that contains this CI, should refer to a universal and individual, which have the CR and IR to which this CI refers.
- This CI refers by-name to a CR and IR. The language definition should contain an MD with matching CR and a PD with matching IR.

AttributeFunction

An *AttributeFunction* (AF) represents an attribution function as discussed in Subsection 2.2.4. It is contained by an IOD. It has several *expressions* to calculate the values in an extensional model. These expressions can refer to the identifiers of the IOD[12]. The AF realizes Need 3.

[11] Future versions of OGML could enforce this by providing a MomentIOD and SubstantialIOD specializing IOD

It has several attributes:

instanceOfRelation, points to the containing IOD

characterizationRole, is a by-name reference to a CI.

name confusingly refers to the property-name of the moment that stores the name of this attribution. This moment of course resides in the extensional model.

lower, holds an expression that should calculate the lower bound of the multiplicity for this attribution. This expression may refer to sequenceIdentifier or the universalIdentifier of the IOD.

upper, holds an expression that should calculate the upper bound of the multiplicity for this attribution. This expression may refer to sequenceIdentifier or the universalIdentifier of the IOD.

naming, holds an expression that should calculate the name for this attribution. This expression may refer to sequenceIdentifier or the universalIdentifier of the IOD.

valuing, holds an expression that should calculate the value for this attribution. This expression may refer to individualIdentifier of the IOD.

typing, holds an expression that should calculate the type for this attribution. This expression may refer to sequenceIdentifier or the universalIdentifier of the IOD.

Constraints:

The CI that this AF refers to by-name should exist (This is a CI with the same value for attribute ***characterizationRole***).

- The lower and the upper expression should return a value of type *integer*.
- The naming expression should return a value of type *string*.
- The valuing expression should return a model constructs from the extensional model.
- The typing expression should return a model constructs from the intensional model.

Abbreviations for the Relational Constructs

Table 4-3 summarizes the abbreviations we used for the *ontological perspective* constructs.

Table 4-3 - Abbreviations for ontological perspective constructs of OGML

Full Name	Abbreviation
InstanceOfDefinition	IOD
AttributeFunction	AF
CharacterizationInstantiation	CI

Instantiation of Ontological Perspective Constructs

Figure 4-9 shows a schematic view of the instantiation semantics of the Ontological Perspective Constructs, in a similar way as in Figure 4-6. IODs are represented as bars with

[12] No assumptions are made about the model query language that is used for the implementation of the expressions. If a query language is not aware of the language axis that OGML introduces in its models, the queries can become quiet complex and hard to understand [27].

the name "IOD". The linguistic instantiation between language and modeling constructs can easily be established (the short downward arrows in the figure)[13]. The Ontological Commitment constructs that were introduced here can help to establish the ontological instantiation shown by the horizontal arrows here.

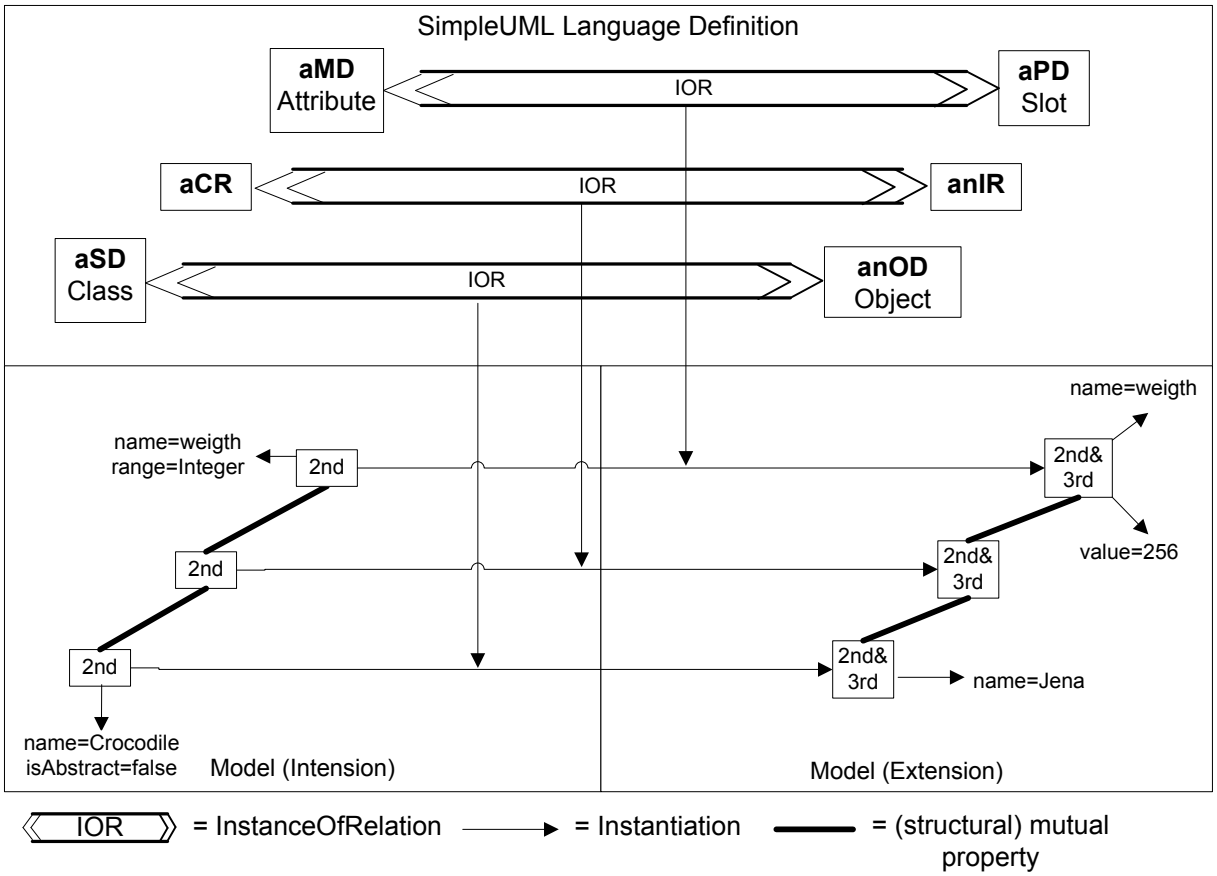


Figure 4-9 - A schematic view of instantiation semantics for OGML relational constructs

Defining the InstanceOfDefinition for SimpleUML

The language SimpleUML needs thus to specify a mapping from the UDs to the IDs in order to make an ontological commitment for the models. Therefore, we define *Relations* between the intensional model constructs and the expressional model constructs. Listing 4-7 shows this. An *IOD* is defined to map the *universals* `Class` on *individuals* `Object` in the model. We can thus distinguish a *moment IOD*, which relates moment universals to moments, and a *substantial IOD*, which relates substantial universals and substantial individuals

The same is done for `Attribute` and `Slot`. Here, we have to be pragmatic to support the demands of modeling. `Classes` cannot be instantiated when marked as abstract in an UML model, even though this violates Ontological laws [30]. Therefore, our *IOD* contains an *instantiation condition* that excludes abstract classes. The *when clause* contains this condition in the form of an OCL expression: **not(**c.isAbstract**)**. In this expression, the variable **c** refers to a Class and is bound to the ***definingConceptIdentifier*** of the IOD. The semantics of the dot in

[13] This will be the task of the modeling tool that is being used for model input

the expression is navigation. Here the structural property `isAbstract` of `Class` **c** is retrieved.

```
Relations UMLInstanceOfAssociationsOnLinks {

        c : Class -> o : Object {
                ...
        } when (not(c.isAbstract))

        a : Attribute -> s : Slot  {
                ...
        }
}
```

Listing 4-7 - OGML by example: defining the InstanceOfRelations

Defining the CharacterizationInstantiation for Substantial IODs

Now that universals are one-to-one mapped to individuals, we can focus on the instantiation of the relations between substantials and non-substantials (moments). Figure 2-2 showed these relations: *characterization* between universals and *inherence* between individuals. Listing 4-8 shows how a *CI* is defined to express the instantiation of the *CR*. Basically, it expresses the fact that the `Attributes` of a `Class` **c** are connected via property `slots` to an `Object` **o**, which is an instance of **c**.

```
Relations UMLInstanceOfAssociationsOnLinks {

        c : Class -> o : Object {
                attributes -> slots;
        } when (not(c.isAbstract))

        ...
}
```

Listing 4-8 - OGML by example: defining the CharacterizationInstantiation

Defining the AttributeFunction for Moment IODs

Here we come to the most interesting part of the OGML language: the definition of the *attribute function* as formalized in [87] and briefly discussed in subsection 0. Listing 4-9 shows the definition of an *AF,* which provides a *perspective* for the moment `Attribute` from the ***point of view*** of a *substantial*. This point of view is established by matching an AF to a CI. An AF is "matched" by the characterizationRole of the CI **ci** according to the pseudo code in Listing 4-10.

```
Relations UMLInstanceOfAssociationsOnLinks {

        ...

        a : Attribute -> s : Slot  {
                attributes {
                        naming name <- a.name;
                        valuing [a.lowerbound .. a.upperbound] s.value;
                        typing a.type;
                }
        }

}
```

Listing 4-9 - OGML by example: defining the AttributionFunction

```
ForAll r in Relations {
        ForAll ior in r.instanceOfRelation {
                ForAll af in ior.attributeFunctions {
                        If (af.characterizationRole = ci.characterizationRole) {
                                Found!
                        }
                }
        }
}
```

Listing 4-10 - Matching of AttributeFunctions with CharacterizationInstantiations

In the previous figues we showed the linguistatic instantiation vertically and ontological instantiation horizontally. For the purpose of explaining the AF we will view the relational constructs as relations orthogonal to the instanceOf relations. Figure 4-10 shows three ortogonal subdivisions between constructs: substantial and moment, universal and individual and language and models forming the three axis: relations (as depth), ontological instanceOf (horizontally) and linguistic instanceOf (vertically).

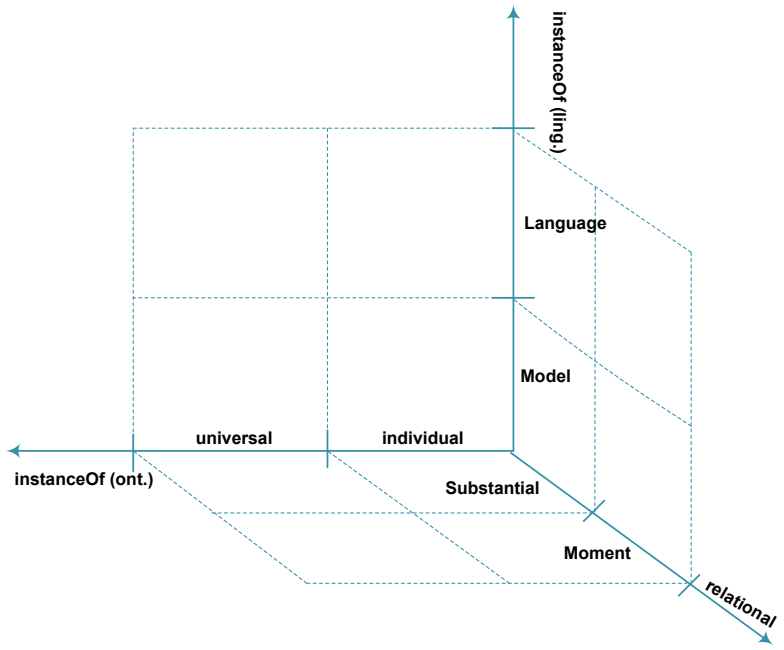


Figure 4-10 - Dimensions in OGML models

To create a view on the participating constructs of in the ontological perspective we populate Figure 4-10. Figure 4-11 shows all the constructs of the example model that are involved in an *ontological perspective* as a cube. A *moment IOD* and a *substantial IOD* are *related* via a *CI*, this is shown in the top of the cube. Their instances are somewhere in the models, but always connected via *explicit instanceOf constructs* represented by the vertical edges of the cube here. Over these instances, the ontological perspective can support *navigation*; *model conformance checking* and *instantiation* of the intensional model (see Need 3).

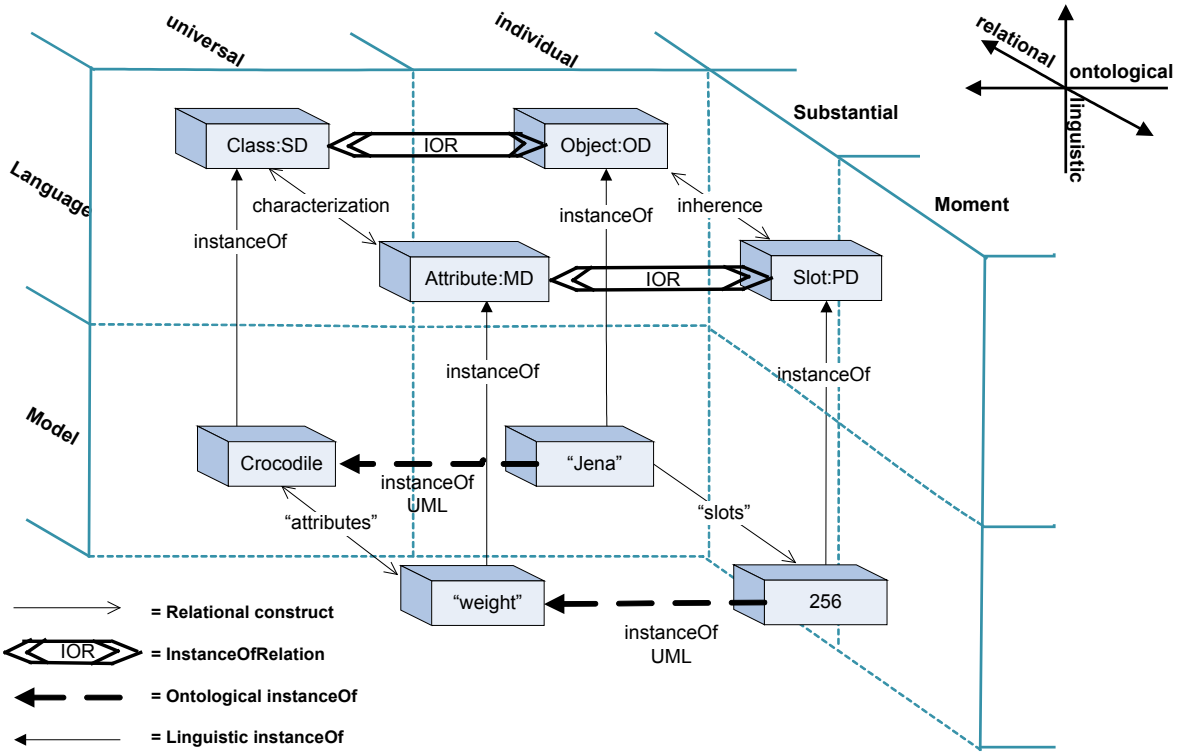


Figure 4-11 - A cross-model view of an established Ontological Perspective

The different expressions of the AF are best understood by analogy with **reflection** in programming languages and metalanguages. A reader that came this far will certainly be familiar with the reflection that EMF provides. In generated *ECore* models, the reflection looks similar to that of Java. An example is shown in Listing 4-11. In the first line the `EObject` is retrieved that represents "Jena" in the model. Subsequently we get its `EClass` and the feature in which we are interested. With the `eGet` function, we can finally retrieve the value of the feature from `jena`. ECore also allows us to retrieve the type of a property with the method `getEType`.

```
EObject jena = getCrocodileFromModel("Jena");
EClass clazz = jena.eClass();
EStructuralFeature feature = clazz.getEStructuralFeature("weight");
Object value = jena.eGet(feature);
Integer weight = (Integer)value;

Class type = feature.getEType().getInstanceClass();

//verify result
if (feature.getUpperBound() > 1 || feature.getUpperBound() == -1) //many
      assert( type instanceof Elist );
else //zero or one
      assert( type.isInstance(weight) );
```

Listing 4-11 – Code example of reflection on an ECore model (EMF)

Similar to the reflection in ECore, we can now query the ontological perspective of the models. However, there is a difference. In ECore models, the reflective functions are present on the model constructs by virtue of inheritance, in OGML they are in the language constructs. Thus, for each query the following steps need to be undertaken:

1. the linguistic instanceOf has to be consulted to retrieve the language constructs,
2. a CI needs to be matched with an AF and,
3. finally some of the AFs expressions need to be executed to retrieve the *wanted value*

The expression that is executed in step 3 can return a model value, because its variables are bound to the model constructs in the bottom of the cube. This binding was described earlier for the *instantiation condition* in Listing 4-7.

Which expression from the AF needs to be executed depends on the operation that is executed on the model and what the *wanted value* actually is. In Table 5, we establish the parallels between the EMF reflection example and the language constructs provided by OGML.

Table 4-4 - Parallels between EMF reflection example and OGML constructs

	ECore	OGML models (bottom cube)	OGML language (top cube)
1	EObject.eClass	ontological instanceOf for the substantial (if stored) or →	following linguistic instanceOf and finding the IOD (if not)
2	EClass.getEStructuralFeature (String)	Linguistic instanceOf and →	***naming*** expression
3	EObject.eGet (EStructuralFeature)	Linguistic instanceOf and →	***valuing*** expression
4	EStructuralFeature.getEType	Linguistic instanceOf and →	***typing*** expression
5	EStructuralFeature.getLowerBound EStructuralFeature.getUpperBound	Linguistic instanceOf and →	***lower*** expression ***upper*** expression

For different purposes: *navigation*, *model conformance checking* and *instantiation*, different combinations of the operations in Table 4-4 can be combined to get the desired result. For example, for navigating the properties of the Crocodile "Jena" we do: **1** → **2** → **3**. Optionally we can execute **4** and **5** to establish the multiplicity and the type of the result.

The Support of More Complex Cases

With an AF we can specify what we see when we look at a *moment* from the perspective of a *substantial*. The example in this section of an `Attribute` is trivial, because an `Attribute` can only be "seen" from the `Class` that is its owner. An Association, however, represents a *mutual property* and can therefore be "seen" from the point of view of multiple substantials. In the next chapter we give a detailed example where we express the *n-ary* **`Associations`** of UML.

4.3.4 Generalization and Specialization Constructs

Generalization is a useful language concept. With it, common properties of a set of substantials can be combined. It can thus reduce redundancy in modeling. The recent MOF specification version 2 [71] and the related UML 2.0 infrastructure specification [74] include property redefinement analogous to what we called "the specialization of properties" in Section 2.2 about Ontology. To constrain the size of this project however, we will not focus on property specialization.

Rather we opt to include the notion of encapsulation. Encapsulation is known in object-oriented paradigms as the hiding of instance variables to minimize the exposure of implementation details to clients [81]. This notion is also adopted in modeling languages like UML where `Attributes` marked as private are not part of the specialization set. Figure 4-12 shows the abstract syntax of the generalization construct.

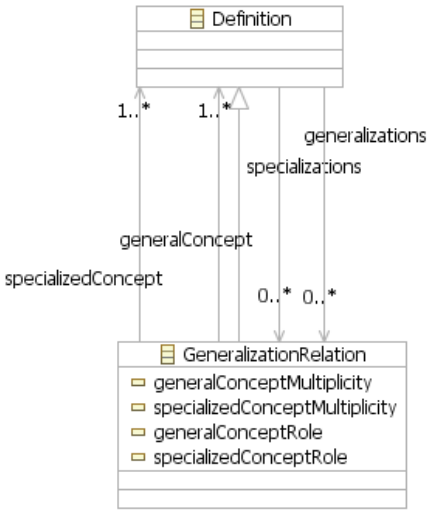


Figure 4-12 - OGML's GeneralizationRelation

GeneralizationRelation

The GeneralizationRelation supports *generalization*. It represents the structural definition of generalization as specified by a language. *SpecializationExclusion* is used to constrain its semantics.

A *GeneralizationRelation* has:

generalConcept, the *Definitions* that support generalization according to the language

specializedConcept, the *Definitions* that support specialization according to the language

generalConceptRole, the property name used to refer to generalizations

specializedConceptRole, the property name used to refer to the specializations of a universal

generalConceptMultiplicity, the multiplicity of generalization assumed by the language

specializedConceptMultiplicity, the multiplicity that the language allows for specialization

Constraints:

- ***generalConcept*** and ***specializedConcept*** may not contain constructs of different ontological categories [39][14]
- ***generalConceptRole*** should not be null
- the ***multiplicities*** may not be zero

SpecializationExclusion

SpecializationExclusion has no equivalent in Ontology. As explained in the beginning of the current section it constitutes to the functionality of object-oriented paradigm as mimicked by modeling languages. The *GeneralizationRelation* cannot directly support the set inclusion semantics for the specializing universals. This is dependent on the different properties that the language defines and therefore is included in *AttributionFunction*. An optional property is added to the *AttributionFunction* as follows:

specializationExclusion, holds an expression that should calculate whether this *moment universal* is part of the set of properties present in a specializing *universal*.

Defining Generalization/Specialization for SimpleUML

In SimpleUML, a `Class` can be *generalized/specialized* from another `Class`. In the example model, the `Crocodile` "inherits" from `Pet`. Without specifying this relation, the `Attribute` "name" of `Pet` remains inaccessible from a `Crocodile`. In Listing 4-12, we see the GeneralizationRelation specified to allow class inheritance.

Encapsulation is supported by specializationExclusion. In *moment IOD* `Attribute` we defined that this moment universal is excluded from the specialization set in the ontological

[14] We do not really enforce this constraint. From the technical perspective, it seems useful to deviate from this constraint from time to time. From ontological perspective this approach is always incorrect because instances of different ontological categories are always disjoint

perspective, when it has a structural property equals to "private". Just like in UML, any attribute marked as private cannot be accessed from instances of subtypes. So if we change the definition of the `Attribute` "name" and give it a value "private" for the attribute visibility, it will become not navigable. Because the specializationExclusion expression will evaluate to *true*.

```
GeneralizationRelation UMLGeneralization {
        generalConcept = Classifier;
        specializedConcept = Class;
        parentMultiplicity = *;
        childMultiplicity = *;
        generalConceptRole = "super";
        specializedConceptRole = "sub";
}

MomentDefinition Attribute {
        ...
        attribute visibility : "String";
        ...
}

Relations UMLInstanceOfAssociationsOnSlots {

        a : Attribute -> s : Slot  {
                attributes {
                        ...
                        specializationExclusion = a.visibility='private';
                }
        }

}
```

Listing 4-12 - OGML by example: defining the inherence for individuals

4.4 Structure of the Modeling Space

OGML is a metalanguage. It can be used to express metamodels, which in turn can express models. In the previous section we have shown how languages (metamodels) can be expressed using the OGML constructs. We did not explain how models are expressed. OGML includes a set of constructs for the expression of models, called OGML eXtensional (OGMLX). These constructs are used *uniformly*, in this case meaning that they are used for normal models as well as metamodels and metametamodels. This makes OGMLX a "model for multiple metalevels" [5] or "modeling space" [62]. How this is realized is explained in subsequent sections. First, we introduce OGMLX.

Figure 4-13 shows the constructs of OGMLX. In the rest of this section, it is explained construct by construct. Constraints are given on its structure that follow from the semantics given in Chapter 6. These constraints narrow down the expressiveness of OGMLX.

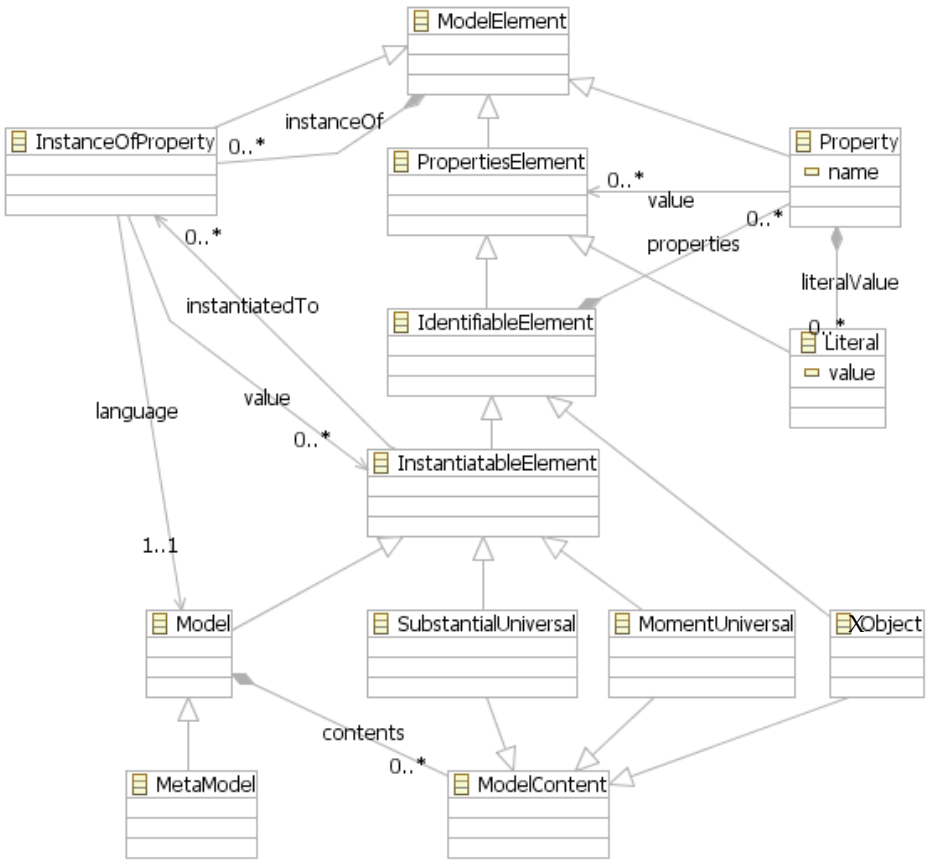


Figure 4-13 - A model for the OGML eXtension (OGMLX)

ModelElement

All OGMLX constructs are model elements (ME). This "top of the lattice" was introduced for the pragmatic purpose of distinguishing extensional constructs from intensional constructs in OGML. It has thus no direct ontological interpretation. For conceptual analysis, the instanceOf property is an attribute of ME. In practice, however the definition of all instanceOf relations would result in recursion ad infinum.

instanceOf, the *InstanceOfProperty* that represents the instantiation of this *ME*.

SubstantialUniversal

A *SubstantialUniversal* (SU) represents a construct in the model that is a *substantial universal* from the point of view of its defining language. It is a specialization of *InstantiatableElement*.

MomentUniversal

A *MomentUniversal* (MU) represents a construct in the model that from the point of view of its metalanguage is a *moment universal*. It is a specialization of *InstantiatableElement*.

XObject

A *XObject* represents a construct in the model that is a *substantial individual* from the point of view of the defining language. It is a specialization of *IdentifiableElement*.

Property

A property represents a construct in the model that is a *moment individual* from the point of view of its defining language. It is a specialization of *PropertiesElement*. Property has several attributes:

name, a unique identifier for this property

value, the values for this property, they must be of type *PropertiesElement.*

InstanceOfProperty

An *InstanceOfProperty* (IOP) is a special property that represents explicitly the membership of a set of properties (a *Kind*) according to BWW or the instantiated to relation in *FCO* [26] (see Figure 2-2). It is a specialization of *ModelElement*. IOP has several attributes:

language, the *MetaModel* of the language that defines this contains the IOD of which this instanceOf is instantiated.

value, the SUs from the *intensional model* (MetaModel) from which the owner of this IOP is instantiated. Although we support multiple values, OGML does currently not support OWL-like multiple instantiation [90].

Literal

Literals represent data values and are related to *intrinsic properties*. Data values have no identity, and thus are not entities according to the dictum of Quine that *"no entities without identity"* [45].

In MOF, for example, values are treated similarly: *"Datavalues act as both Instances (since they may have slots) and as ValueSpecifications: since datavalues are always considered directly stored in a slot rather than being referred to (which would require some sort of identity)"* [71].

In BWW [87]: values are elements of the codomains of attribute functions. They cannot exist independently in the world. Instead, they must be conceived in terms of things that have properties that in turn are represented as values of attribute functions.

The current version of OGML does not provide extensive support for literals. For simplicity, all data values are represented by untyped *Literals* and stored as `String`. It will be *future work* to provide a full and correct interpretation of literals. A Literal has:

value, the String value of the data

IdentifiableElement

An *IdentifiableElement* (IE) represents all constructs in the model that from the point of view of the metalanguage is *substantial* and thus can have properties. It is a specialization of *PropertiesElement*. An IE has:

properties, the properties of an object give it identity. Contrary to the ontological point of view, the values of properties are not the direct representation of identity [90]. In information systems, each object already has identity through its memory location. Common practice is to assign a canonical value to this identity in the

form of a variable name or key attribute. Currently OGML assumes the "*name*" property as identification. It will be future work to make this language-independent by providing an identification function for definitions.

Constraints:

- All properties of an *IE* **ie** that must be *unique* with respect to the *language* that defines their instanceOf and their ***name*** (see Equation 1). Identical names for properties are thus allowed because the associated language definitions provide a different ontological perspective on the model.
- Each *IE* **ie** must have an instanceOf according to OGML (**ogml**), because models and languages are both instanceOf the metalanguage (see Equation 2). We will investigate this claim in subsequent sections.
- All *IE* **ie** must have an instanceOf in OGML, because models and languages are instanceOf the metalanguage and for all substantials, we record this instanceOf relation as was explained in the introduction of the current section.

$$[\forall p_1 : Property, \nexists p_2 : Property \mid p_1 \epsilon\ ie.properties \wedge p_2 \epsilon\ ie.properties \wedge [\exists iop_1 : InstanceOfProperty, \exists iop_2 : InstanceOfProperty \mid p_1.instanceOf = iop_1 \wedge p_2.instanceOf = iop_2 \wedge iop_1.language = iop_2.language] \wedge p_1.namep_2.name \wedge p_1 \neq p_2] \quad : \textbf{Equation 1}$$

$$[\exists iop : InstanceOfProperty \mid iop \epsilon\ ie.instanceOf \wedge iop.language = ogml] \quad : \textbf{Equation 2}$$

MetaModel

A *MetaModel* (MM) can contain a set of constructs and can be used as intensional model[15]. It is a specialization of *Model*.

Constraints:

- All metamodels contain at least one universal (see Equation 3)

$$\forall mm : MetaModel[\exists c : InstantiatableElement[c \in mm.contents\ ]] \quad \textbf{: Equation 3}$$

Model

A *Model* can contain a set of constructs and can be used as extensional model[15]. It is a specialization of *IdentifiableElement*.

contents, the constructs that are included in this model

Constraints:

- All constructs within one model are instantiated according to the same languages as the model itself (see Equation 4).

[15] Model constructs should be viewed as incompletely specified. It will be future work to introduce the containment relation between models and their constructs and to give a useful interpretation to the *LanguageDefinition*. The result will be a redefinition of the Model and MetaModel construct in OGMLX

$\forall m: Model[\forall i_1: InstanceOfProperty[\forall e_1: IdentifiableElement[i_1 \in m.instanceOf \wedge e_1 \in m.contents \rightarrow (\exists i_2: InstanceOfProperty[i_1.language = i_2.language]]]]$: **Equation 4**

The Ontological Ground of OGMLX

In Table 4-5, we provide the ontological grounding of OGMLX. The construct equivalence comes from Guizzardi [43]. Properties are expressed handled differently in BWW; we provide some references for their interpretation in the table.

Table 4-5 - Abbreviations and ontological equivalents for constructs of OGML eXtensional

Full Name	Ontological equivalent		Abbre-viation
	FCO	BWW ontology	
SubstantialUniversal	*Substantial universal*	*Kind* (*Functional Schema* [30])	SU
XObject	*Substantial individual*	*Thing*	-
MomentUniversal	*Moment universal*	*Attribute*	MU
Property	*Moment individual*	*Property*	-
ModelElement	all	all	ME
InstantiatableElement	*Universal*	-	-
IdentifiableElement	*Substantial*	-	IE
PropertiesElement	all (in absence of an *IOP*)	all except *Property* [87]	PE
InstanceOfProperty	formal relation [26]	membership [39]	IOP
Literal	element in *set of values* [43]	element in set of values [87]	-
Model	a model / a world / an extension		-
MetaModel	an ontology / an intension		MM

Representing Models

Figure 4-14 shows the example Crocodile models in OGMLX form. Arrows represent properties and dashed arrows represent instanceOf properties. This figure does not extend the complete metamodel of SimpleUML. A reader could imagine how "OGML" instanceOf properties link the boxes to their language definitions. In this case: `Class`, `Attribute`, `Object` and `Slot`. The following chapters will explain in more detail how all models are instantiated to OGMLX.

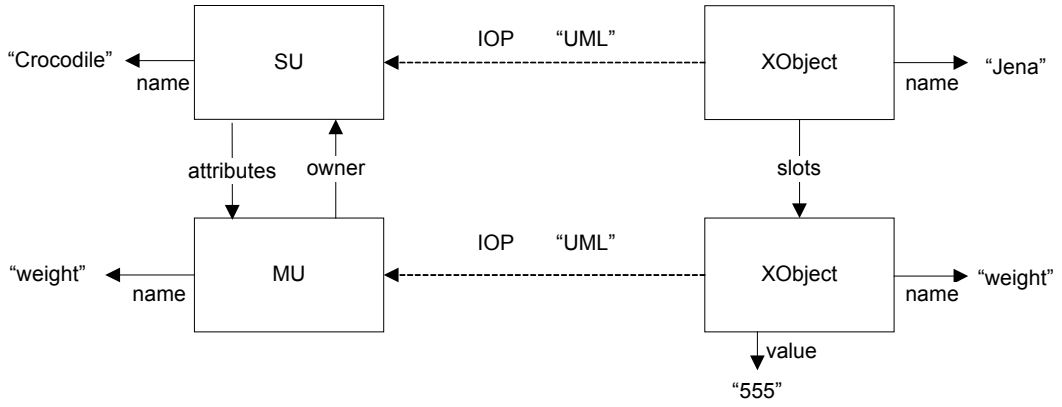


Figure 4-14 - An example OGMLX model

4.5 How OGML is Self- Reflective

In the section 4.3, we introduced OGML by applying it in an example. In this example, we expressed a modeling language in the OGML metalanguage. From this point of view, OGML is indeed a metalanguage. However, in Section 2.3 we explained how a language can be represented by its metamodel and thus is itself a model. From this point of view, OGML is also a modeling language and can thus express itself (see Figure 4-15). In addition, by doing so, we can show to a certain extend that OGML is expressive enough for the domain of modeling languages and their instantiation semantics. In the current section, we reify the OGML constructs.

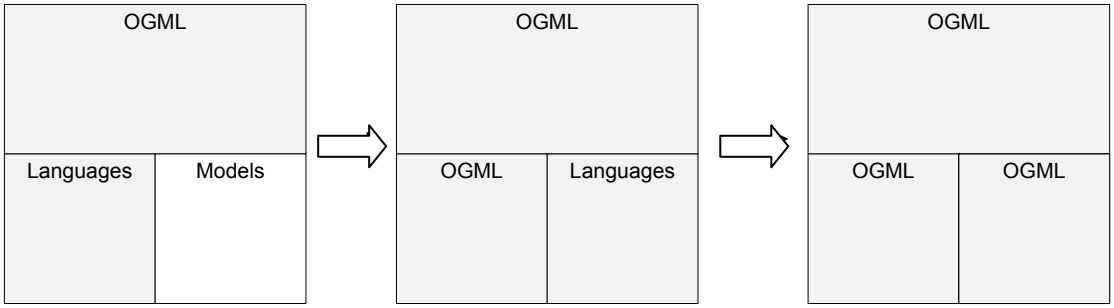


Figure 4-15 - OGML as a modeling language defined by itself

In Subsection 2.3.4, we explained that a language should make a choice between expressiveness and preciseness and that no language can express its own semantics. A language can represent its own abstract syntax, using its own concrete syntax in order to support semantics definition, but this always needs to be supplemented with a semantics description, where the constructs are mapped on an existing semantics domain. This mapping and the formalization of the semantics we provide in Chapter 6. In the current section, we present the self-describing nature of OGML, which can be seen as an addition to the more informal semantics description found in Section 4.3.

The full definition of OGML is found in Appendix B. The concrete syntax used here is shown in Appendix A.

Language Constructs

Listing 4-13 shows that the language constructs are all represented as SDs. From the point of view of Ontology, even a definition of a moment individual is a substantial, because as we saw in Figure 4-3 it can be instantiated in the model. The generalization relations between these constructs form the subdivision of ontological categories as seen in Figure 2-2 and Figure 4-2.

Class and *OGMLDataType* are introduced here for the first time here. They are used for the language constructs that are not instantiated to the models. Class is used for *Expressions*, which only exists in language models. The *OGMLDataType* is used to define properties with data value (intrinsic properties) in de language definition, as the "*name*" of Definition. String, Integer and other data types are defined from line 80. There are other constructs, related to language semantics that do not have to be instantiated in the models, like AFs and CIs. These can all be represented using classes (see lines 165 to 185 and 225 to 231).

```
2:      SubstantialDefinition Definition extends Classifier {
                attribute name : "String";
        }
5:      SubstantialDefinition "SubstantialDefinition" extends UniversalDefinition {}
        SubstantialDefinition "MomentDefinition" extends UniversalDefinition {}
        SubstantialDefinition "DataTypeDefinition" extends UniversalDefinition {}

10:     SubstantialDefinition IndividualDefinition extends Definition {}
        SubstantialDefinition "ObjectDefinition" extends IndividualDefinition {}
        SubstantialDefinition "PropertyDefinition" extends IndividualDefinition {}

        SubstantialDefinition "Class" extends Definition {}
15:     DataTypeDefinition "OGMLDataType" extends Definition {}

        SubstantialDefinition LanguageDefinition {
                attribute definitions [*] : Definition, "Relations", "GeneralizationRelation";
        }

        ...

80:     Class "OclAny" {}
        OGMLDataType "String" extends "OclAny" {}
        OGMLDataType "Integer" extends "Double" {}
        OGMLDataType "Boolean" extends "OclAny" {}
        OGMLDataType "Double" extends "OclAny" {}
```

Listing 4-13 - The OGML definition of Language Constructs

A SD is instantiated three times, as we saw in Figure 4-3. The instances become universal definitions, which can be instantiated again. The SD behaves like a *"Clabject"*. This can be explained by looking at Figure 4-16. The individual is individual because it is instantiated in three times from the SD. However, the same can be said about the *substantial universal* in the figure, because the SD is self-defined. It is thus also an individual. In this manner, *OGML defines this Clabject nature of constructs in a consistent way.* Depending on the language point of view (examples can be found in Figure 3-6, Figure 3-7 and Figure 4-15), one part of the properties will be "seen" as *structural* and another part as *ontological*. Just as described in Subsection 4.3.3.

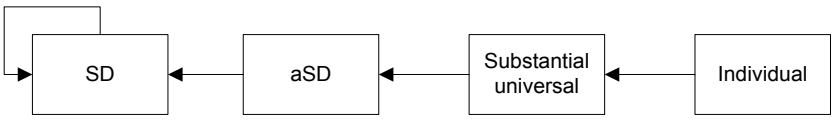


Figure 4-16 - Instantiation of a SubstantialDefinition

Attribute

Line 21 to 27 (see Listing 4-14) shows the *Attribute* being represented as a MD. An Attribute itself has attributes (instances of Attribute) for *name*, *type range* and *multiplicity*. Careful inspection of the syntax (see the *range* attribute instance on line 23) shows that all of these attributes can be given values. Lastly, an Attribute is connected to a Definition representing the attribution. From the point of view of Ontology, the MD Attribute characterizes the Definition. The CR is shown on line 26.

```
21:     MomentDefinition Attribute {
                attribute name : "String";
                attribute range  [1-*] : Definition;
                attribute lower : "Integer";
25:             attribute upper : "Integer";
                characterization " owner " : Definition momentDefinitionRole [*] " attributes ";
        }
```

Listing 4-14 - The OGML definition of Attribute

Characterization

Line 29 to 37 show the CR (see Listing 4-15). Like an Attribute, a characterization is also a MD. The difference is that it characterizes two Definitions, a MD and a UD. Lines 35 and 36 show these characterizations. The CR on line 35 characterizes the MD. According to the semantics of CR it will thus become navigable from a MD via a property called *"dependency"*. From the CR we can reach the MD via *"dependentDefinition"*. For pragmatgical reasons OGML is currently limited to *binary CRs*. We show however in the next chapter that this is sufficient to express n-ary relations.

```
31:     MomentDefinition CharacterizationRelation {
30:
                attribute lower : "Integer";
                attribute upper : "Integer";
                attribute "momentDefinitionRole" : "String";
                attribute "universalDefinitionRole" : "String";
35:             characterization dependentDefinition : MomentDefinition momentDefinitionRole [1-*]  dependency;
                characterization ownerDefinition : UniversalDefinition momentDefinitionRole [*]  feature;
        }
```

Listing 4-15 - The OGML definition of CharacterizationRelation

It should be noted that line 35 and 36 express the relation between the CR construct and the MD and SD constructs again using CRs (instances of CR constructs). Characterization becomes a recursive concept in this manner. To understand the meaning of these relations we are referred back again to the CR construct, conceptually we find an infinite number of small characterizations. This is shown in Figure 4-17 using UML `Object`, `Class` and `Association` notation to express three steps of this recursion. For pragmatic reasons, tools will have to hardcode the interpretation of the CR.

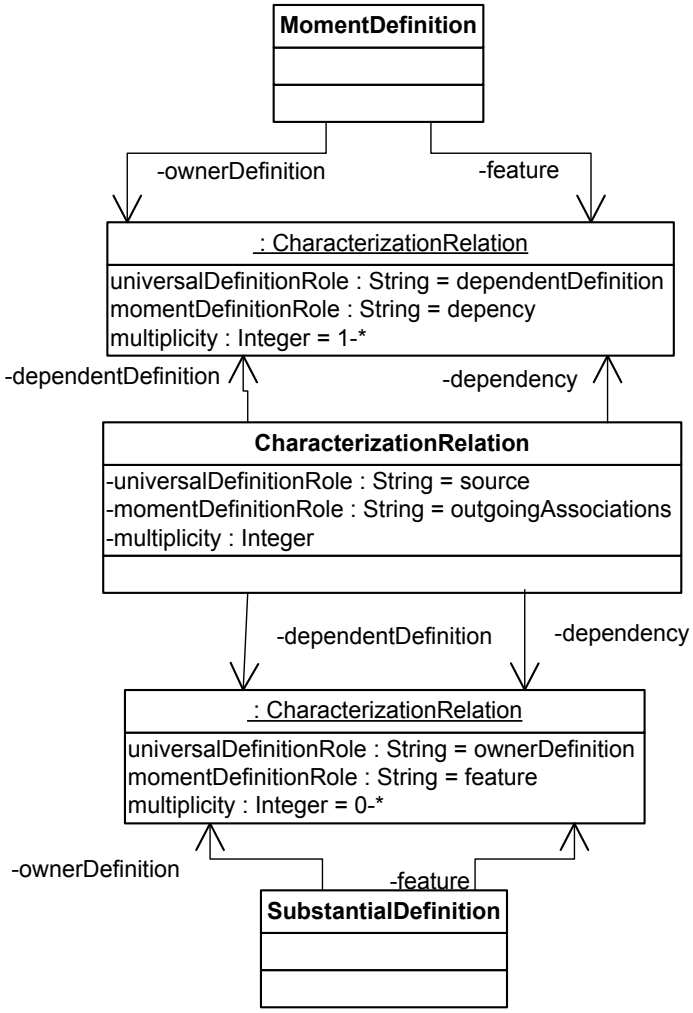


Figure 4-17 - The recursive nature of the CharacterizationRelation

The Modeling Space

OGML also provides a structure to express models on; it is called OGMLX, and was introduced in the previous section. In the same fashion that OGML constructs are used to express other languages, OGMLX constructs are also represented using OGML constructs. This is shown in Appendix B on lines 189 to 222 and in Listing 4-16. Intentionally the constructs and relations between them are exactly the same as in Figure 4-13. *OGMLX thus becomes a linguistic instanceOf OGML.*

```
        SubstantialDefinition ModelElement {}
190:    SubstantialDefinition PropertiesElement extends ModelElement {}
        SubstantialDefinition IdentifiableElement extends PropertiesElement {}
        SubstantialDefinition InstantiatableElement extends IdentifiableElement {
                attribute instantiatedTo[*] : InstanceOfProperty;
        }
195:    SubstantialDefinition ModelContent extends IdentifiableElement {
                attribute container : ModelContent;
        }

        SubstantialDefinition "Model" extends IdentifiableElement {
200:            attribute contents [*] : ModelContent;
        }
        SubstantialDefinition "MetaModel" extends "Model", InstantiatableElement {}

        SubstantialDefinition "SubstantialUniversal" extends InstantiatableElement, ModelContent {}
205:    SubstantialDefinition "MomentUniversal" extends InstantiatableElement, ModelContent  {}

        ObjectDefinition XObject extends IdentifiableElement, ModelContent {}
        ObjectDefinition Literal extends PropertiesElement {
                attribute "value" [*] : "String";
210:    }

        PropertyDefinition Property extends ModelElement {
                attribute name : "String";
                attribute "value"[*] : PropertiesElement;
215:            dependsOn IdentifiableElement role = "properties" multiplicity = * ;
        }

        PropertyDefinition InstanceOfProperty extends ModelElement {
                attribute "value"[*] : "Model", "MomentUniversal", SubstantialUniversal;
220:            attribute "language" : "String";
                dependsOn PropertiesElement, Property, InstanceOfProperty role = "instanceOf" multiplicity = *;
        }
```

Listing 4-16 - The OGML definition of OGMLX constructs

The OGML InstanceOfDefinitions

The OGML definition also contains IODs. They relate all OGML constructs to OGMLX constructs. The full definition is found in Appendix B on lines 87 to 163. An example of the IODs is shown in Listing 4-17. It shows how the SD and the MD are mapped to SU and MU. The CR is mapped to two properties, each for one direction of the relation, so one will be attached to a MD and another to a SD.

```
Relations OGMLInstanceOfDefinition {
        ...
        "SubstantialDefinition" -> SubstantialUniversal {}
         sd : UniversalDefinition -> su : InstantiatableElement {
                feature -> properties;
        }
        md : "MomentDefinition" -> mu : MomentUniversal {
                dependency -> properties;
        }

        c : CharacterizationRelation -> p1 : Property {
                feature {
                        naming name <- c."momentDefinitionRole";
                        valuing [c.lower .. c.upper] p1.value;
                        typing c.dependentDefinition;
                }
        }
        c : CharacterizationRelation -> p2 : Property {
                dependency {
                        naming name <- c."universalDefinitionRole";
                        valuing [0 .. -1] p2.value;
                        typing c.ownerDefinition;
                }
        }
        ...
}
```

Listing 4-17 - The OGML definition of OGMLX constructs

By defining the IODs in this manner we recognize that *OGML* as metalanguage *provides instanceOf semantics between languages and models* as we saw in Figure 3-6. *By defining these instanceOf semantics between OGML and OGMLX constructs, we express the linguistic instanceOf between models and languages*. In effect, we treat *linguistic instanceOf* the same as the *ontological instanceOf*. Depending on the choice of language perspective; OGML or a modeling language, different IODs are recognized. Thus, either the instanceOf between languages and models (linguistic) is "seen" or the instanceOf between models (ontological).

In the next subsection, we will show how all constructs of OGML, languages expressed in OGML and their models are mapped on OGMLX (the modeling space).

4.6 How OGML is Mapped to the Modeling Space

In section 4.4, we introduced a structure for the modeling space, which we call *OGML eXtensional (OGMLX)*. OGMLX is used to *uniformly represent* models and metamodels (including OGML itself). To achieve this all *modeling and language constructs* have to be uniquely projected on the OGMLX structure for the whole modeling architecture. In the current section, we show how this *structural reification* is realized. We do this with a *proof*. First, we reiterate the *base facts* for the proof.

Intension – Extension Dichotomy

The first basis to realize *uniform representation* is done by making OGMLX part of OGML. This was explained in the previous section. The OGML language definition thus contains two kinds of constructs as shown in Figure 4-18.

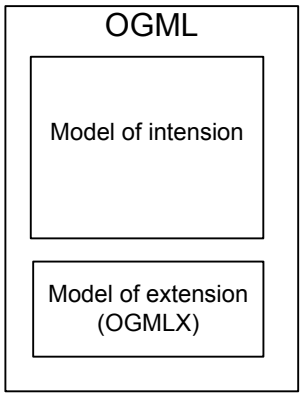


Figure 4-18 - OGML definition divided in a model for intension and extension

Three sets of constructs are thus found in OGML:

- OGML contains *OGML constructs,*
- OGML contains an intensional model, which contains *intensional constructs,*
- OGML contains OGMLX, which contains *OGMLX constructs.*

The latter two sets of constructs are thus disjoint and complete subsets of the first set.

InstanceOf Semantics between Languages and Models

The second basis is the definition of IODs in OGML; they relate all OGML constructs to OGMLX constructs. This was partly described in the previous section. A summary of all the IODs in the OGML definition is shown in Listing 4-18. The ***definingConcepts*** on the left are all OGML constructs and the ***conformingConcepts*** on the right are all OGMLX constructs. The relations between them are expressed by CIs and AFs, which are not shown in the listing. The full definition is found on lines 87 to 163.

```
Relations OGMLInstanceOfRelation {

        abstract Definition -> PropertiesElement {... }
        sd : UniversalDefinition -> su : InstantiatableElement {... }
        md : "MomentDefinition" -> mu : MomentUniversal {... }
        "PropertyDefinition" -> XObject {}
        "SubstantialDefinition" -> SubstantialUniversal {}
        "DataTypeDefinition" -> SubstantialUniversal {}
        "ObjectDefinition" -> XObject {}
        "Class" -> XObject {}
        "OGMLDataType" -> Literal {}
        InstanceOfRelation -> InstanceOfProperty {}

        ld : LanguageDefinition -> mm : MetaModel {... }

        a : Attribute -> p : Property {... }

        i : InherenceRelation -> p : Property {... }

        c : CharacterizationRelation -> p1 : Property {... }
        c : CharacterizationRelation -> p2 : Property {... }

        g : "GeneralizationRelation" -> p1 : Property {... }
        g : "GeneralizationRelation" -> p2 : Property {... }
}
```

Listing 4-18 - The OGML definition of OGMLX constructs

Basis of the Proof

Now we show how the OGML definition and semantics ensure that all modeling constructs have a representation on OGMLX. To proof this, we formulate the facts. Based on the following four facts about OGML we can make an inference:

Fact 1: By definition OGMLX is a linguistic instance of the intensional model (**first basis**)
Fact 2: By definition the intensional model is a linguistic instance of itself (**first basis**)
Fact 3: The existence of an IOD for each construct in the intensional model (**second basis**)
Fact 4: The semantics of the IOD (see subsection 4.3.3)

The proof presented here consists of *four steps*. In Step 1**,** we will prove that OGML constructs are indeed instances of OGMLX. Thereafter in Step 2**,** we do the same thing for language constructs from languages other than OGML. Then we show in Step 3 how model constructs become an instance of OGMLX. Finally, we generalize the proof to a statement about all modeling constructs in Step 4 (in models, languages and OGML). Natural deduction is used to make the inference. To save space a shorthand notation for logic predicates is used; n-ary predicates are written with capital letters and are immediately followed by variables. To give an example: *instanceOf(a, b)* is represented as *IOab* .

Step 1: Proof for OGML Constructs

Here we proof that *OGML constructs* are *instance of OGMLX*. We assume the following *predicates* to distinguish between different constructs in OGML:

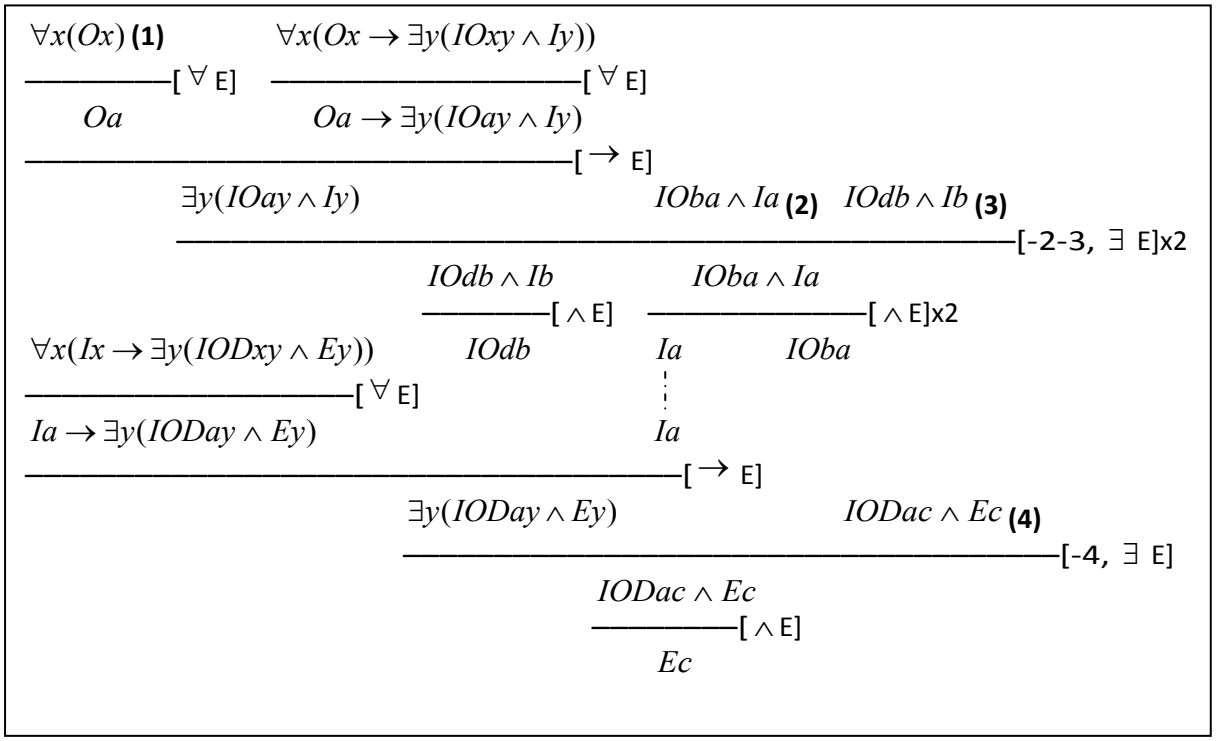
$Ox \leftrightarrow$ x ∈ OGML (x is an intensional construct or an OGMLX construct)
$Ex \leftrightarrow$ x ∈ OGMLX (x is a OGMLX construct)
$Ix \leftrightarrow$ x ∈ OGML\OGMLX (x is an intensional construct)

We recognize the following 2-ary predicates for relations:
$IOxy \leftrightarrow x$ is an instance of y (from OGML point of view we do not distinguish *ontological* or *linguistic*)
$IODxy \leftrightarrow$ there is an IOD inside the OGML definition relating x to y

Premises:
We represent Facts 1 and 2 in first-order logic:
Premise 1: $\forall x(Ox \rightarrow \exists y(IOxy \wedge Iy))$
We represent Fact 3 in first-order logic:
Premise 2: $\forall x(Ix \rightarrow \exists y(IODxy \wedge Ey))$
We represent Fact 4 in first-order logic:
Premise 3: $\forall x \forall y(Ix \wedge Ey \wedge IODxy \rightarrow \forall x' \forall y'(IOy'x' \wedge IOx'x \rightarrow IOy'y))$

To proof:
All OGML constructs are instance of OGMLX, in first-order logic:
$\forall x(Ox) \rightarrow \forall x \exists y(IOxy \wedge Ey)$

Proof:
First, we make a trivial deduction of formulas with valid combinations of constants. Listing 4-19 shows how we derive four lemmas from the premises and assumption $\forall x(Ox)$ **(1)**.



Listing 4-19 - Proof of a set of base formulas deduced from the premises (Step 1a)

Under assumption **(1)** we thus found the following formulas to be true: $\underline{IOdb \wedge Ib}$, $\underline{IOdb}$, $\underline{IOba \wedge Ia}$, $\underline{IOba}$, $\underline{Ia}$, $\underline{IODac \wedge Ec}$ and $\underline{Ec}$. These and assumption **(1)** will be used in the deduction shown in Listing 4-20.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x \forall y(Ix \wedge Ey \wedge IODxy \rightarrow \forall x' \forall y'(IOy'x' \wedge IOx'x \rightarrow IOy'y)) \quad Ia \quad IODac \wedge Ec}{Ia \wedge Ec \wedge IODac \rightarrow (\forall x' \forall y'(IOy'x' \wedge IOx'a \rightarrow IOy'c)} [2x \forall E] \quad \dfrac{}{Ia \wedge IODac \wedge Ec} [\wedge I]}{\forall x' \forall y'(IOy'x' \wedge IOx'a \rightarrow IOy'c)} [\rightarrow E] \quad \dfrac{IOdb \quad IOba}{IOdb \wedge IOba} [2x \forall E] \quad [\wedge I]}{IOdb \wedge IOba \rightarrow IOdc} [\rightarrow E] \quad \dfrac{Ec \qquad IOdc}{IOdc \wedge Ec} [\wedge I]}{\exists y(IOdy \wedge Ey)} [\exists I]}{\forall x \exists y(IOxy \wedge Ey)} [\forall I]}{\forall x(Ox) \rightarrow \forall x \exists y(IOxy \wedge Ey)} [-1, \rightarrow I] \quad \blacksquare$$

Listing 4-20 - Proof that OGML constructs are part of OGMLX (Step 1b)

This proves that Facts 1-4 ensure that every OGML construct is instance of the extension.

Step 2: Proof for Language Constructs

Now we can prove also that the constructs of *other languages* also become instance of the OGMLX as a result of Facts 1-4. We introduce a new set of constructs:
$Lx \leftrightarrow x \in$ Language\OGML (*x* is a construct in any language, not including OGML)

Premises:
And we use the fact that all language definitions are a linguistic instance of the OGML intension, expressed in first-order logic:
Premise 4: $\forall x(Lx \rightarrow \exists y(IOxy \wedge Iy))$

To proof:
All language constructs are instance of OGMLX, in first-order logic:
$\forall x(Lx) \rightarrow \forall x \exists y(IOxy \wedge Ey)$

Proof:
This proof now does not look different from the previous proof. We only have to exchange Premise 1 for Premise 4 and assumption **(1)** from $\forall x(Lx)$ instead of $\forall x(Ox)$. As a direct result of changing assumption **(1)** the conclusion becomes $\forall x(Lx) \rightarrow \forall x \exists y(IOxy \wedge Ey)$. This shows that language constructs are part of the OGMLX.

Step 3: Proof for Model Constructs

Here we prove *models* to be part of OGMLX. The proof is similar to the proof presented in Step 1, but here we have to make slightly different assumptions. Again, we introduce a new set of constructs:

$Mx \leftrightarrow x \in \text{Model}$ (x is a construct in a model)

Premises:

We keep all the premises from the earlier proof and add a premise that expresses the fact that models are linguistic instances of modeling languages (model constructs are linguistic instances of the set of Language constructs). This is expressed with the following formula in first-order logic:

Premise 5: $\forall x(Mx \rightarrow \exists y(IOxy \wedge Ly))$

To proof:

All model constructs are instance of OGMLX, in first-order logic:

$\forall x(Mx) \rightarrow \forall x \exists y(IOxy \wedge Ey)$

Proof:

Like in Step 1, we first make a trivial deduction of formulas with valid combinations of constants. Listing 4-21 shows how we derive four lemmas from the premises and assumption $\forall x(Mx)$ **(1)**.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x(Mx)\ \textbf{(1)}}{Md}[\forall E] \quad \dfrac{\forall x(Mx \rightarrow \exists y(IOxy \wedge Ly))}{Ma \rightarrow \exists y(IOay \wedge Ly)}[\forall E]}{\exists y(IOay \wedge Ly)}[\rightarrow E] \quad IOba \wedge La\ \textbf{(2)} \quad IOdb \wedge Lb\ \textbf{(3)}}{}[\text{-2-3, } \exists \text{ E}]\text{x2}}{IOba \wedge La}[\wedge E] \quad \dfrac{IOdb \wedge Lb}{La}[\wedge E]\text{x2}}{IOdb}}{\forall x(Lx \rightarrow \exists y(IOxy \wedge Iy))}[\forall E] \quad \vdots}{La \rightarrow \exists y(IODay \wedge Ey)} \quad La}[\rightarrow E]}{\exists y(IODay \wedge Ey)} \quad IODac \wedge Ec\ \textbf{(4)}}{}[\text{-4, } \exists \text{ E}]}{IODac \wedge Ec}[\wedge E]}{Ec}
$$

Listing 4-21 - Proof of a set of base formulas deduced from the premises (Step 3a)

Under assumption **(1)** we thus found the following formulas to be true: $\underline{IOdb \wedge Lb}$, $\underline{IOdb}$, $\underline{IOba \wedge La}$, $\underline{IOba}$, $\underline{La}$, $\underline{IODac \wedge Ec}$ and $\underline{Ec}$. These and assumption **(1)** will be used in the deduction shown in Listing 4-22.

$$\dfrac{\dfrac{\dfrac{\dfrac{\forall x\forall y(Ix \wedge Ey \wedge IODxy \rightarrow \forall x'\forall y'(IOy'x' \wedge IOx'x \rightarrow IOy'y))}{Ia \wedge Ec \wedge IODac \rightarrow (\forall x'\forall y'(IOy'x' \wedge IOx'a \rightarrow IOy'c)} [2x\forall E] \quad \dfrac{IODac \wedge Ec \qquad Ia}{Ia \wedge IODac \wedge Ec} [\wedge E]}{\forall x'\forall y'(IOy'x' \wedge IOx'a \rightarrow IOy'c)} [\rightarrow E]}{IOdb \wedge IOba \rightarrow IOdc} [2x\forall E] \quad \dfrac{IOdb \quad IOba}{IOdb \wedge IOba} [\wedge I]}{IOdc} [\rightarrow E]$$

$$\dfrac{\dfrac{\dfrac{\dfrac{Ec \qquad IOdc}{IOdc \wedge Ec} [\wedge I]}{\exists y(IOdy \wedge Ey)} [\exists I]}{\forall x\exists y(IOxy \wedge Ey)} [\forall I]}{\forall x(Mx) \rightarrow \forall x\exists y(IOxy \wedge Ey)} [-1, \rightarrow I] \blacksquare$$

Listing 4-22 - Proof that model constructs are instances of OGMLX (Step 3b)

From $\forall x(Mx) \rightarrow \forall x\exists y(IOxy \wedge Ey)$ we can conclude that model constructs are instance of the OGMLX.

Step 4: Generalization of the Proofs

In the current section, we have proven the *adequateness* with which the OGML modeling architecture represents the constructs from *models*, *languages* and *OGML* itself. This indeed includes all constructs *C* that are present in the OGML architecture: $\forall x(Cx \rightarrow Ox \vee Lx \vee Mx)$ thus:

$$\dfrac{\dfrac{\dfrac{\dfrac{\forall x(Cx \rightarrow Ox \vee Lx \vee Mx)}{Ca \rightarrow Oa \vee La \vee Ma} [\forall E] \quad \dfrac{\forall x(Cx)\ \textbf{(4)}}{Ca} [\forall E]}{Oa \vee La \vee Ma} [\rightarrow E] \quad \dfrac{\dfrac{\forall x(Ox) \rightarrow \forall x\exists y(IOxy \wedge Ey)}{Oa \rightarrow \forall x\exists y(IOay \wedge Ey)} [\forall E] \quad \begin{matrix}\ldots & \ldots & \ldots \\ \textbf{(1)} & \textbf{(2)} & \textbf{(3)} \\ Oa & \ldots La & \ldots Ma\end{matrix}}{\forall x\exists y(IOay \wedge Ey)} 3x[\rightarrow E]}{\forall x\exists y(IOay \wedge Ey)} [-1\ -2\ -3, \vee E]}{\forall x(Cx) \rightarrow \forall x\exists y(IOxy \wedge Ey)} [-4, \rightarrow I] \blacksquare$$

Listing 4-23 - A generalization of the proofs in step 1, 2 and 3 (Step 4)

The presented proof is *adequate* in the sense that it merely proves *existence* of an instanceOf OGMLX. A *uniqueness* prove is not directly given, but follows from the fact that the instanceOf relations are functions. This intuition becomes clearer when we represent the proofs as a conceptual graph [85]. Figure 4-19 shows such a *conceptual graph*. The boxes represent quantification over the sets of constructs that we defined. These are related to each

other via the 2-ary predicates IO and IOD (shown as ovals in the figure). The IO relations that we inferred in the current section are shown with dashed lines.

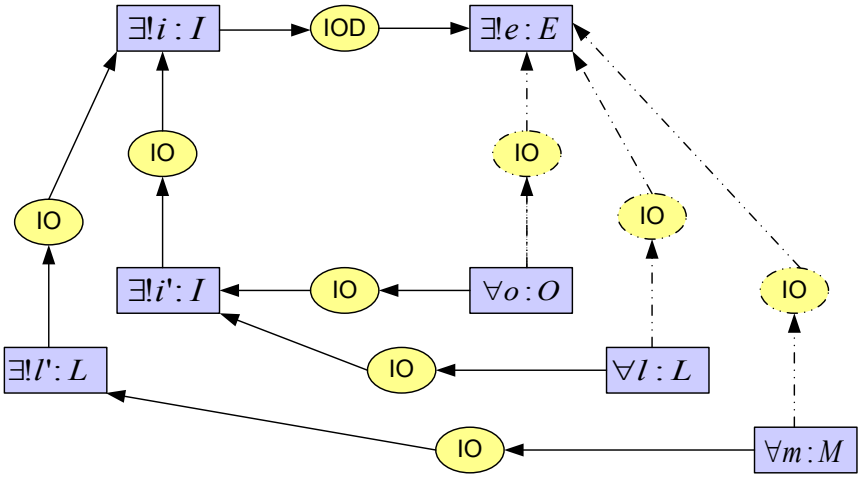


Figure 4-19 - A conceptual graph of the model constructs mapped to OGMLX

Of particular interest are the *O L* and *M* sets, which represent OGML, Language and Model constructs. For all elements ***x*** in those sets the instanceOf *exactly one* ***e*** has been established as follows:

The OGML definition ensures that each ***x*** is always related to exactly one *i* via two *IO* relations. That this is exactly one *i* is shown by the uniqueness quantors (∃!) and follows from the fact that instanceOf relations are functions. An IOD relation relates this *i* to one *e,* which by semantics of IOD now becomes the defining constructs for ***x***.

4.7 The Resulting Modeling Architecture

Literature on metamodeling provides detailed comparisons between the different design options for modeling architectures [11][34][10][3][9]. Some of which have been explained in Section 2.4 and Chapter 3. In the current section, we use some of these design options to evaluate the properties of the OGML architecture.

Modeling Architecture

In the previous section, we showed how from the point of view of the metalanguage, OGML, all model constructs are instances of OGMLX. OGMLX is at the same time part of the OGML definition. Figure 4-20 shows the architecture of OGML by grouping the sets f constructs that exist in the architecture. It shows a nested architecture like discussed in subsection 2.4.7. This architecture resembles the one resented by Kurtev in [62] (Section 2.5). This observation will be revisited in Chapter 8.

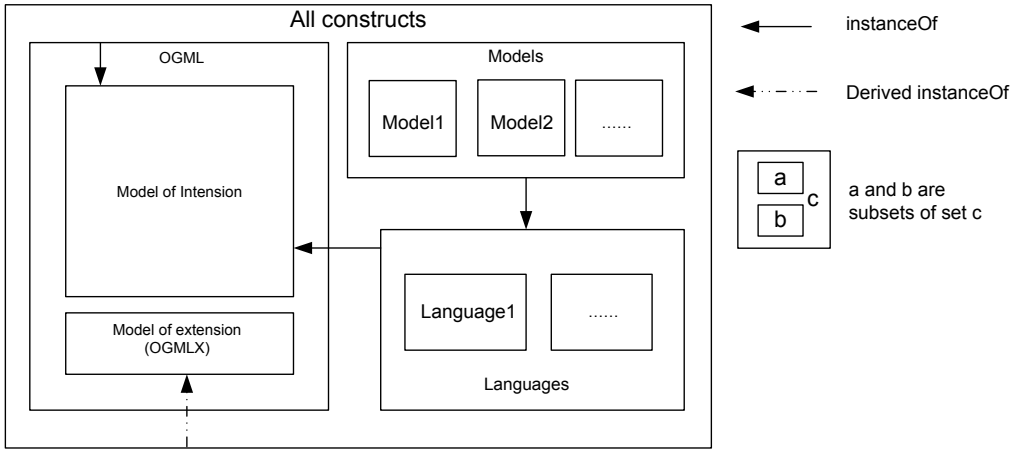


Figure 4-20 - The OGML architecture represented as sets of constructs

Instantiation Semantics

Gitzel and Hildebrand reason about *uniform* and *layer dependent instantiation semantics* [34]. They do not discuss the possibility to achieve both. This is what OGML has achieved by making the instantiation semantics part of the metamodel definition. OGML realizes the "M2-level mechanism" that Atkinson and Kühne discuss [10].

The Number of Layers

We assume three modeling layers counting all the models as one layer; no fourth layer exists (see Figure 4-21). We argue that this is a truthful representation of what actually exist in the modeling architecture from the point of view of the metalanguage.. The description of power types all happens on the model layer and if the language definition supports this.

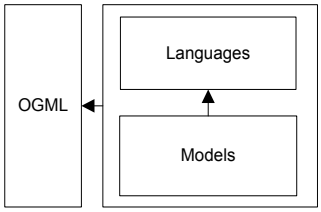


Figure 4-21 - The nested modeling architecture of OGML

Recursive or Axiomatic Metalanguage

We did not choose for an axiomatic definition of the metalanguage. The reflective definition provides the ability to reuse the same principle over multiple layers. An example is the support of "*Clabject*" like constructs on all layers, which is needed for the pragmatics of modeling; on all layers languages and models need to be specified, thus constructs need to be named and additional information needs to be attached for notational purposes. The last sections of the current chapter have shown how OGML supports this in a uniform manner over all layers.

Self-reflective systems do not exist according to Gödels theorem. Whereas a conceptual language can define its own constructs, using its own constructs, finally it needs to be mapped on some existing structure to be stored in. For OGML this is OGMLX. OGMLX

ultimately needs to be expressed on some existing structure, like a model in *ECore*. This we have done in the prototype, which is presented in Chapter 7. Conceptually OGML constructs reify their own definition. However, the semantics still need to be expressed by a mapping to graph structures in Chapter 6 and are supported by the OCL semantics.

4.8 Conclusions

In the current chapter, we proposed a metalanguage based on Ontology and with *explicit instanceOf relation*. We drew *design decisions from the domain of Formal Ontology* and presented the resulting metalanguage complete with syntax, semantics description and associated modeling architecture.

We introduced OGML by means of a running example. We showed how *an ontological view* on the models *supports metamodeling decisions.* At first hand it may seem that define languages in OGML is a complex undertaking. However, consider how a metamodeler would have dealt with these precise metamodeling tasks in, for example, a MOF environment: all universals on all layers would be `Classes`, likewise for attributes and there is no way to specify semantics of the languages. There is simply no way how a modeler could end up with favorable features as "language independent model handling" in a MOF metamodeling.

We also showed how the OGML constructs support a structural perspective on models as well as the ontological perspective. The ontological perspective is realized by the language definition in OGML. In section 4.5, we described OGML's self-reflective nature. Here we were able to conclude that *OGML handles the "Clabject" nature of constructs in a consistent way over all layers.*

Finally, we derived conclusions from the resulting architecture. In section 4.6, we proved how OGML projects all modeling constructs on a fixed structure called OGMLX. For OGML itself this structure is seen as the extension where all models reside. A tool that implements OGML, however, can map the structure directly onto a data structure and thus has a *uniform representation of models, language models and the OGML model itself.*

As Kurtev showed in [62] (Section 2.5) a traditional architecture, like MOF, cannot explicitly represent this relation to the underlying structure. This may be caused by the fact that MOF's original purpose was to be a data-exchange format, later it evolved to a metalanguage: *"Although … MOF has its origins as … supporting model interchange … it quickly became associated with the meta-metamodel at level M3. Unfortunately, these two interpretations are not compatible in the original linear metamodeling framework."* [7],

Section 4.7 shows the OGML architecture in terms of the relations between its constructs. To summarize in the terminology of Atkinson and Kühne [11]: this shows OGML to be a *nested* modeling architecture with *three layers* and a *compaction level* formed by OGMLX. OGMLX is in their terms a *library format* for all models and languages and *also* a *language format* from the point of view of OGML.

Chapter 5 – Case Studies

5.1 Introduction

In the previous chapter, OGML was introduced. We did this by means of a simple example of an UML model with Classes and Attributes. In the current chapter, we extend the UML example with associations. To show the capabilities of OGML, we do this in different ways. Afterwards we draw conclusions about the differences in metamodeling choices.

All the examples presented in the current chapter have been verified by a prototype implementation of OGML, which is presented in Chapter 7. We extensively use UML and OCL here, so a reader is supposed to be familiar with their specifications [75][73].

About UML

UML is a vast modeling language that focuses on different aspects of software design. It includes modeling languages to model behavior well as structure. We will focus only on a structural (classes) part of the language and in particular on the relational features there: *Association*, *Inheritance* and *Attribution*. The UML specification [75] explains how associations are instantiated to links and attributes to slots.

We expressed two flavors of UML called**:** *SimpleUML1* and *SimpleUML2*. While the two do not differ significantly in the constructs they provide (both focus on attributes and associations) their intention is different. The purpose of *SimpleUML1* is to show the proposed metalanguage can define different mechanisms for "UML Object" model instantiation. The *SimpleUML1* definition contains three different definitions for the instantiation semantics: associations instantiated to binary links, navigable associations with attributes (association classes) and association instantiated to slots.

SimpleUML2 focuses on the expression of *n-ary* links [36]. We show that the result is a navigable model. Both adaptations of UML provided us with insights in the ontological nature of the UML constructs, their differences make explicit the design choices that have to be made when designing a modeling language.

5.2 SimpleUML1

In SimpleUML1, we define three different views on the world. They differ in the relation individual they define: binary links, attributable links and only slots. Listing 5-1 shows the universal definitions we define in this language and Listing 5-2 shows the individual definitions we define in this language. We will use the same ontological commitment of the language to demonstrate three different instantiation mechanisms.

```
SubstantialDefinition UMLClass {
        attribute name : "String";
        attribute isAbstract : "Boolean";
}

GeneralizationRelation UMLGeneralization {
        generalConcept = UMLClass;
        specializedConcept = UMLClass;
        parentMultiplicity = *;
        childMultiplicity = *;
        generalConceptRole = "super";
        specializedConceptRole = "sub";
}

OGMLDataType "String" {}
OGMLDataType "Boolean" {}

DataTypeDefinition UMLPrimitiveType  {
        attribute name : "String";
}

MomentDefinition UMLAttribute {
        attribute name : "String";
        attribute upper : "String";
        attribute lower : "String";
        attribute visibility : "String";
        attribute type : UMLClass, UMLPrimitiveType;
        attribution universalDefinition = UMLClass, UMLAssociation
                                universalDefinitionRole = "owner"
                                momentDefinitionRole = "attributes"
                                multiplicity = *;
}

MomentDefinition UMLAssociation {
        attribute name [0-1] : "String";
        attribute sourceClassRole : "String";
        attribute targetClassRole : "String";
        attribute sourceClassLower : "String";
        attribute sourceClassUpper : "String";
        attribute targetClassLower : "String";
        attribute targetClassUpper : "String";
        outgoing universalDefinition = UMLClass universalDefinitionRole = "source"
                                momentDefinitionRole = "outgoingAssociations" multiplicity = *;
        incoming universalDefinition = UMLClass universalDefinitionRole = "target"
                                momentDefinitionRole = "incomingAssociations"  multiplicity = *;
}
```

Listing 5-1 - Case study: SimpleUML1, universal definitions

```
ObjectDefinition UMLObject {}
ObjectDefinition UMLLiteral {
        attribute value : "String";
}

PropertyDefinition UMLSlot {
        attribute name : "String";
        attribute value : UMLObject, UMLLiteral;
        dependsOn UMLObject, UMLLink role = "slots" multiplicity = *;
}

PropertyDefinition UMLLink {
        attribute sourceObject : UMLObject;
        attribute targetObject : UMLObject;
        attribute sourceRole : "String";
        atribute targetRole : "String";
        dependsOn UMLObject role = "outgoingLinks" multiplicity = *;
        dependsOn UMLObject role = "incomingLinks" multiplicity = *;
}
```

Listing 5-2 - Case study: SimpleUML1, individual definitions

5.2.1 Associations of Binary Links

In UML, binary associations are navigable. For each pair of objects that is associated on link is created. The definition of the `UMLAttribute` is similar to the one defined in the previous chapter. It contains a specializationExclusion, to support encapsulation of attributes. A new feature shown here is the realization of the `UMLAssociation`. The UML specification defines that its instances are links and states the following:

Instantiation – *"An association declares that there can be links between instances of the associated types. A link is a tuple with one value for each end of the association, where each value is an instance of the type of the end."*

Navigation – *"The function roles(as) = <r1, . . . , rn> assigns each class ci for 1 < i < n participating in the association a unique role name ri…"*

Therefore, for each combination of combined model elements a link is created. An association is thus instantiated to an unknown number of links. Therefore, the ***sequenceIdentifier*** of the IOD is used. Line 19 of Listing 5-3 shows this (links : [l : UMLLink]). Navigation over binary associations can start in an `UMLClass x` and ends in the associated UML`Class y`. An `UMLClass` can be instantiated to several `UMLObjects`. If the multiplicity in this direction is higher than one, the result of the navigation on the instances can be a set of `UMLObjects`. Navigation in two directions are supported by two AFs defined on lines 20 and 25.

```
1:      Relations UMLInstanceOfSimple {
            t : UMLPrimitiveType -> l : UMLLiteral {}

            c : UMLClass -> o : UMLObject {
5:              attributes -> slots;
                outgoingAssociations -> outgoingLinks;
                incomingAssociations -> incomingLinks;
            } when (not(c.isAbstract='true'))

10:         a : UMLAttribute -> s : UMLSlot  {
                attributes {
                    naming name <- a.name;
                    valuing [a.lower .. a.upper] s.value;
                    typing a.type;
15:                 specializationExclusion = a.visibility='private';
                }
            }

            a : UMLAssociation -> links : [l : UMLLink] {
20:             outgoingAssociations {
                    naming targetRole <- a.targetClassRole;
                    valuing [a.targetClassLower .. a.targetClassUpper] links.collect(l | l.targetObject);
                    typing a.target;
                }
25:             incomingAssociations {
                    naming sourceRole <- a.sourceClassRole;
                    valuing [a.sourceClassLower .. a.sourceClassUpper] links.collect(l | l.sourceObject);
                    typing a.source;
                }
30:         }
        }
```

Listing 5-3 - Case study: SimpleUML1, binary link instantiation

We investigate the instantiation of relation *outgoingAssociations*. Its ***valuing*** expression is on line 22 (at the end) and uses the variable *links*, which is bound to all links instantiated from

the specific association. The result of this expression is thus a collection of all the endpoints of these links. Only those are selected that conform to the naming condition on line 21: their "targetRole" property needs to have a value equal to the "targetClassRole" property of the `UMLAssociation` *a* (a.targetClassRole). Together with the results from ***typing*** and ***multiplicities***, we defined the navigation result that the UML and OCL specification prescribe.

An Example Model with Associations

Figure 5-1 shows the instantiation of the `UMLAssociations` in with UML notation. The top of the figure shows the *Class Language* definition of SimpleUML1, with an SD for `UMLClass`, an MD for the `UMLAssocciation` and two CRs to connect them. At the bottom, the *Object Language* definition is shown, with an OD for the `UMLObject`, a PD for the `UMLLink` and one IR drawn as association. The middle of the figure shows a *Class Diagram* for the gynecology of crocodiles ("In UML" box). InstanceOf relations are drawn with dashed arrows. In The *Object Diagram* model, we see the double instantiations of all constructs.

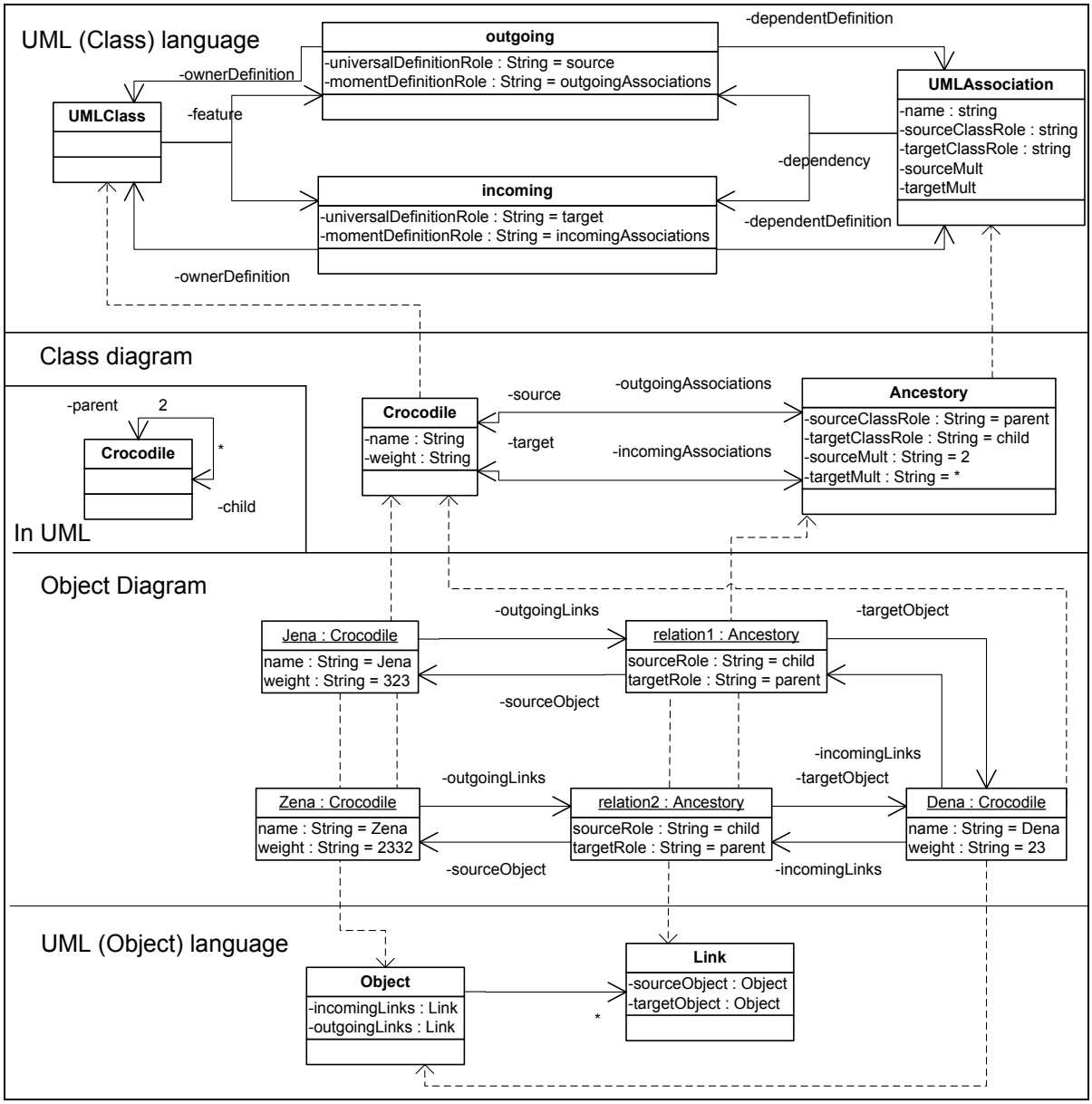


Figure 5-1 – SimpleUML1 models in OGML with instantiated associations

A Demonstration of Navigation

As an example, we navigate the ontological perspective of the models using the IODs of Listing 5-3. The exercise will be to find all association ends of crocodile "Dena". This is an `UMLObject` instanceOf an `UMLClass`. Therefore, we bind the identifiers of the IOD on line 6 to the `UMLClass` and `UMLObject` in the models:

o = `Dena`
c = `Crocodile`

At the Class Diagram level, we can establish the type of the association end and the multiplicity. The IOD for Crocodile (line 6) has two CIs. We take *outgoingAssociations* and navigate (structurally) to find the model constructs there: c.outgoingAssociations. We find `Ancestory`; an `UMLAssociation`. We bind it to *a* of the IOD at line 19. We can thus find a perspective for the CI in the IOD for `UMLAssociation`, provided that it has an AF that matches the CI. Such a one is found at line 20. We can execute different expressions from the AF now:

naming, a.targetClassRole = "child"
typing, a.target = `Crocodile`
lower, a.targetClassLower = 0
upper, a.targetClassUpper = *infinite*

We can the same thing for *incomingAssociations* and find:
naming = "parent",
typing = `Crocodile`,
lower = 2 and
upper = 2

On the instance level, a similar thing can be done to get the link ends. We start again with *outgoingAssociations* and navigate (structurally) to find the model constructs there: o.outgoingLinks. An empty set is the expected result: "Dena" has no children. This conforms to the found multiplicity for *outgoingAssociations*. Next we try *incomingAssociations.* We find `relation1` and `relation2`, both `UMLLinks`. We bind them as a *set* to *links* of the IOD at line 19. Now we can execute the valuing function:

valuing, links.collect(l | l.targetObject) = {"Dena", "Zena"} (conforms to the found multiplicity of 2)

We found that "Dena" (*o*) is connected to two other UMLClasses:

o.child = {} of type `Crocodile`
o.parent = {"Dena", "Zena"} of type `Crocodile`

5.2.2 Associations on Attributable Links (AssociationClass)

Here we define SimpleUML1 with the concept of attributable associations: `AssociationClasses`. For SimpleUML1 we do not follow the specification, but rather stick to the ontological commitment that SimpleUML1 already makes. SimpleUML2 will give an example that is closer to the specification.

A reader who took a close look at the ontological commitment shown in Listing 5-1 and Listing 5-2 may already have noticed that attributable associations and links are already structurally supported by it. The MD `UMLAttribute` characterizes both `UMLClass` and `UMLAssociation` and PD `UMLSlot` inheres in `UMLObject` and `UMLLink`.

When an association has attributes, these need to be accessible. In the previous subsection, the navigation passed the association and ended directly in the opposite class. In this example, we make it end in the association itself. From there, the attributes can be queried or alternatively one can navigate to one of the association endpoints. Listing 5-4 shows how the IOD for `UMLAssociation` has a CI to access the attributes (see line 8). The AFs from the previous example are also present with a small modification to make the navigation end in the association itself (the end of lines 12 and 17).

Thus far, we can navigate to associations and access its attributes there, but no perspective is provided on the endpoints of the association itself. Therefore, we define the CIs on line 5 and 6. They use the CRs, which characterize `UMLClass` with `UMLAssociation`, the other way around: from association to class and thus refer to the *universalDefinition* property of the CRs. To implement the perspective on classes we need AFs for these CIs.

To define these *AFs* we have to do something asymmetric. One would expect that these are added to the IOD for `UMLClass`. However, a SD contains no information on attribution, this is the role of the MD. Therefore, completely asymmetric with the previous AFs we specify, we add the *AFs* in the same IOD for `UMLAssociation`. Line 21 and 26 shows them. Their multiplicity is always one, because if a link exists, we know it always links two classes.

```
2:      Relations UMLInstanceOfWithLinksAsAssociationClass {
            ...
            a : UMLAssociation -> links : [l : UMLLink] {
5:              source -> sourceObject;
                target -> targetObject;

                attributes -> slots;

10:             outgoingAssociations {
                    naming targetRole <- a.targetClassRole;
                    valuing [a.targetClassLower .. a.targetClassUpper] links;
                    typing a;
                }
15:             incomingAssociations {
                    naming sourceRole <- a.sourceClassRole;
                    valuing [a.sourceClassLower .. a.sourceClassUpper] links;
                    typing a;
                }
20:
                source {
                    naming sourceRole <- a.sourceClassRole;
                    valuing [1 .. 1] links.collect(l | l.sourceObject);
                    typing a.source;
25:             }
                target {
                    naming targetRole <- a.targetClassRole;
                    valuing [1 .. 1] links.collect(l | l.targetObject);
                    typing a.target;
30:             }
            }
        }
```

Listing 5-4 - Case study: SimpleUML1, binary link instantiation with properties

5.2.3 Associations on Slots

The previous examples showed how `UMLAssociation` is mapped on an `UMLLink`. According to BWW, an association is nothing more than a set of mutual properties [87]. Here we define the instantiation of the binary association on a pair of slots. Listing 5-5 shows the definition of two IODs for `UMLAssociation`.

```
Relations UMLInstanceOfAssociationsOnSlots {
        ...

        a : UMLAssociation -> s1 : UMLSlot {
                outgoingAssociations {
                        naming name <- a.sourceClassRole;
                        valuing [a.sourceClassLower .. a.sourceClassUpper] s1.value;
                        typing a.source;
                }

        }, a : UMLAssociation -> s2 : UMLSlot {
                incomingAssociations {
                        naming name <- a.targetClassRole;
                        valuing [a.targetClassLower .. a.targetClassUpper] s2.value;
                        typing a.target;
                }
        }
}
```

Listing 5-5 - Case study: SimpleUML1, slot instantiation

5.3 SimpleUML2

SimpleUML2 defines *n-ary* associations with properties, which are navigable.

In UML, Associations do not per se have to be navigable. Only binary Associations without properties (AssociationClasses) are navigable and only when marked explicitly as navigable. In the SimpleUML1 example, we already assumed all ends navigable in order to demonstrate the capabilities of OGML. In SimpleUML2, we will define *n-ary* associations with properties, which are navigable. Instantiation of n-ary properties in UML happens just like joining several tables in database systems. In databases for each combination of rows, a new row is created, in UML a link is created for each combination of objects participating in the Association. The specification says the following about n-ary associations:

"For n-ary associations, the lower multiplicity of an end is typically 0. A lower multiplicity for an end of an n-ary association of 1 (or more) implies that one link (or more) must exist for every possible combination of values for the other ends."

In SimpleUML1, the `Association` was defined using one *MD* and two *CRs*. For SimpleUML2 we cannot do this because the number of associated classes can be *n* while the *CR* is always binary (see subsection 4.3.2). Thus to support n-ary associations, we make a new ontological commitment. Listing 5-6 shows the universal definition in this commitment, Listing 5-7 shows the individuals. `Association` is defined on an *SD* and we added an *MD* `AssociationEnd` to represent the association ends. For the individuals this translates to an *OD* `Link` and a *PD* `LinkEnd`. The new commitment allows us to specify the `AssociationClass` explicitly while in SimpleUML1 every `UMLAssociation` is an `UMLAssociationClass`. `AssociationClass` extends `Class` and `Association` just like in the UML specification.

```
SubstantialDefinition Classifier {
        attribute name : "String";
}

SubstantialDefinition "Class" extends Classifier {
        attribute isAbstract : "Boolean";
}

MomentDefinition Attribute {
        attribute name : "String";
        attribute lowerbound : "String";
        attribute upperbound : "String";
        attribute type : Classifier;
        attribution universalDefinition = "Class" universalDefinitionRole = "owner"
                        momentDefinitionRole = "attributes" multiplicity = 1-*;
}

SubstantialDefinition Association extends Classifier {}

SubstantialDefinition AssociationClass extends "Class", Association {}

MomentDefinition AssociationEnd {
        attribute roleName [0-1] : "String";
        attribute lowerbound : "String";
        attribute upperbound : "String";
        outgoing universalDefinition = "Class" universalDefinitionRole = "class"
                                momentDefinitionRole = "associations" multiplicity = *;
        incoming universalDefinition = "Association" universalDefinitionRole = "association"
                                momentDefinitionRole = "memberEnds" multiplicity = 2-*;
}
```

Listing 5-6 - Case study: SimpleUML2, universal definitions

```
ObjectDefinition Link {}

PropertyDefinition LinkEnd {
        attribute object : Object;
        attribute roleName : "String";
        attribute link : Link;
        dependsOn Object role = "links" multiplicity = *;
        dependsOn Link role = "ends" multiplicity = 2-*;
}
...
```

Listing 5-7 - Case study: SimpleUML2, individual definitions

The IODs are defined in Listing 5-8. The IOD for `AssociationClass` inherits the CIs and AFs from `Class` and `Association` and needs no additions, thus is empty. The IOD for `Class` specifies CIs for attributes and associations. The IOD for `Association` contains a CI for `AssociationEnds`. It can be instantiated to multiple `Links` therefore a ***sequenceIdentifier*** is used (links : [I : Link]).

`AssociationEnds` are thus seen from the point of view of `Classes` and for the point of view of `Associations`. Therefore, they have two AFs. The first one ("associations") implements the point of view from the Class. Its lengthiness stems from the fact that the number of instantiated links can vary widely. The lower bound ***lb*** of this instantiation multiplicity can be established by multiplying all the lower bounds of each individual `AssociationEnd`. From the point of view of one class attached to an `AssociationEnd` *ae*, we "see" a minimum of *lb* / *ae*.lowerbound `AssociationEnds`. Expressed by the ***lower*** expression (see line 17 - 18):

```
ae.association->first().memberEnds->select(roleName<>ae.roleName)->iterate(end ; lower : Integer = 1 |
                                                                    lower * end.lowerbound.toInteger())
```

The ***upper*** expression of "associations" AF follows with the same reasoning, only we have to account for **-1** as being *infinite*. It is shown on lines 20 to 26. The "memberEnds" *AF* definition is simpler reflecting the fact that each link is always connected to one object at each link end. The lower bound multiplicity can still be zero, because not all `Classes` need to participate in n-ary links. In binary links obviously, they have to. Otherwise, there would be no link.

```
1:  Relations SimpleUML2Instance {
        AssociationClass -> objects : [o : Object] {}

        c : "Class" -> o : Object {
5:              attributes -> slots;
                associations -> links;
        } when (not(c.isAbstract='true'))

        a : Association -> links : [l : Link] {
10:             memberEnds -> ends;
        }

        ae : AssociationEnd -> ends : [le : LinkEnd] {
                associations {
15:                     naming roleName <- ae.roleName;
                        valuing [
                                ae.association->first().memberEnds->select(roleName<>ae.roleName)
                                ->iterate(end ; lower : Integer = 1 | lower * end.lowerbound.toInteger())
                            ..
20:                             let uppers : Collection(Integer) = ae.association->first().memberEnds
                                ->select(roleName<>ae.roleName)[upperbound.toInteger()] in
                                if (uppers->includes(-1)) then
                                    -1
                                else
25:                                 uppers->iterate(u ; upper : Integer = 1 | upper * u)
                                endif
                            ]
                            ends.collect(e | e.link);
                        typing ae.association;
30:             }

                memberEnds {
                        naming roleName <- ae.roleName;
                        valuing [ae.lowerbound..1] ends.collect(e | e.object)->first();
35:                     typing ae.class;
                } where (ends->size()=1)
        }
    }
```

Listing 5-8 - Case study: SimpleUML2, instantiation

5.4 Querying the Models

In the current section we show some example queries that can be run on models conforming the languages that we defined in Chapters 4 and 5. The queries run on our prototype OCL interpreter. The inputs are the query, the models, the language model and the OGML model. All are represented in OGMLX form as we will show in Chapter 7. To support the different perspectives that OGML provides, the OCL syntax was expanded with a languageAxis construct. This allows a user to explicitly state according to which language the navigation over the models should happen. An example of this expression is:

```
{SimpleUML1 | jena.children }
```

This switches to the ontological view that SimpleUML1 provides. Take for example the crocodile model. From the point of view of UML, we can query a specific crocodile, say Jena, for its children. If we query an UML model with the following OCL query: jena.children, we expect a set of children crocodiles returned. A user of UML is not at all interested that the model is represented as Objects and slots on the mechanical level. And querying the object model in OGML terms would require a query like: jena.slots->select(s | s.name='children').collect(value). It is clear that a language without this ontological perspective would be quiet useless.

Another non-standard notation is adopted from the OCL Interpreter in ATL [52]. In order to separate the OCL metamodel from the ATL transformation language metamodel, ATL proposes a construct to refer to model elements by thieir model name and element name in the following manner: UMLMM!Player

Query on 4-ary Associations

To demonstrate the OGML navigation with the OCL Interpreter, we created the model in Figure 5-2. It contains a *3-ary association with AssociationClass*. Navigation for `Classes` in SimpleUML2 is defined to go to the `Association`(`Class`) . However we cannot easily use the opposite end label as we did for SimpleUML1, where jena.child->collect(c | c.child) resulted in all the children of jena. Here the label of the own link end is used: player.player results in the set of all link ends connected to player.

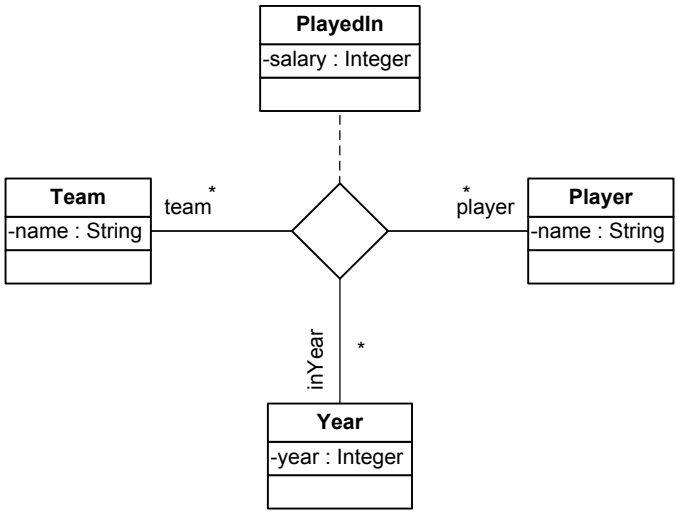


Figure 5-2 - An example UML model with a 3-ary association and an AssociationClass

In Table 5-1, the contents are shown for a model conforming to the model shown in Figure 5-2 and to SimpleUML2.

Table 5-1 - The model contents for a model conforming to the model in Figure 5-2

Player	Team	Year	salary
Davids	TWENTE	1999	1000000
Kluivert	TWENTE	2000	100000
Davids	AJAX	-	200000
Kluivert	AJAX	1998	500000

An OCL query is defined in Listing 5-9. It uses standard OCL notation [73]. The LanguageAxis expression is used to use the (ontological) navigation provided by the SimpleUML2language. The query consists out of iterations: over all players *ps* and for each *ps* over all their AssociationClasses *ac*. The ends of each ac are used to compose a String.

```
{ SimpleUML2 | UMLMM!Player.allInstances()->collect(ps | ps.player->collect(ac |
        'Player '+ ac.player.name + ' played in team '+ ac.team.name +
        if ac.inYear.oclIsUndefined() then
                        ''
                else
                        ' during '+ inYear.year
                endif
        + ' for the mere sum of $'+ ac.salary))
}->iterate(row ; result: String = '' | result + row + '\n')
```

Listing 5-9 - Case study: an example OCL query on SimpleUML2 models

The resulting String of executing this query is shown in Listing 5-10.

```
Player Davids played in team TWENTE during 1999 for the mere sum of $1000000
Player Kluivert played in team TWENTE during 2000 for the mere sum of $100000
Player Davids played in team AJAX for the mere sum of $200000
Player Kluivert played in team AJAX during 1998 for the mere sum of $500000
```

Listing 5-10 - Case study: the results of the query in Listing 5-9

Figure 5-3 shows an instance model containing one link of one football player (the second link is connected via the line that runs to the bottom). The inter-model arrows represent the instanceOf relations. Using this model we investigate how language-independent querying is realized using the LanguageAxis expression.

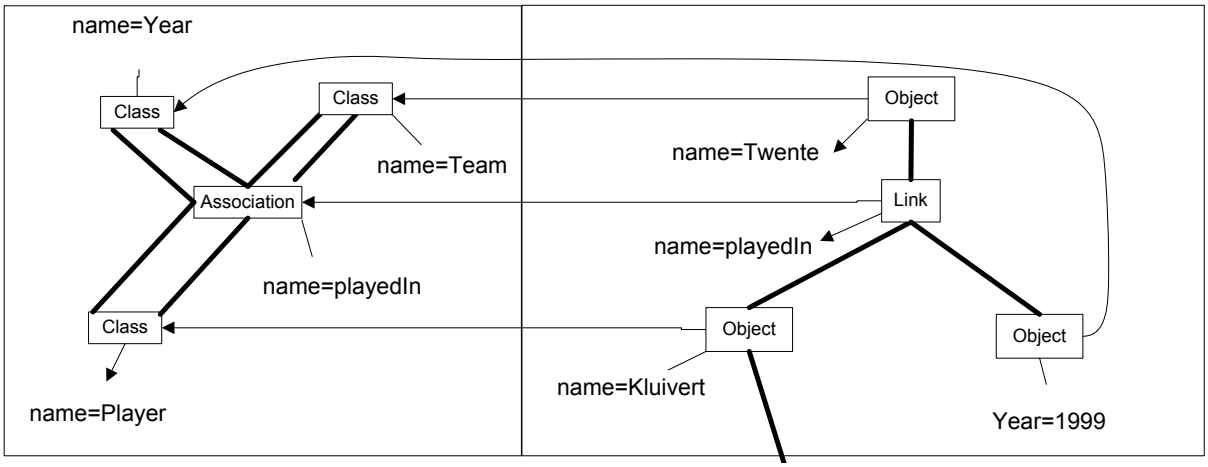


Figure 5-3 - An example instance model with associations

The same model could also be queried using two other language perspectives. Linguistically:

```
{ OGML | UMLModel!Kluivert.links }
```

This will return the `LinkEnds` that are attached to player "Kluivert". The expression UMLModel!Kluivert is used here to return the model element named "Kluivert". Structuturally the model can also be queried (OGMLX):

```
{ OGMLX | UMLModel!Kluivert.properties }
```

This will return the IRs that connect player "Kluivert" to its - seemingly not so directly attached - `LinkEnds`. The language axis chosen here is OGMLX, although one would expect it to be OGML, because it defines this instanceOf relation. In Sections 4.6 and 4.7, we witnessed how all constructs become instanceOf OGMLX by OGML definition. All languages thus became linguistic instanceOf OGML (intension) and instanceOf OGMLX. In the prototype, we therefore have to distinguish between the two instanceOf relations. Therefore, whenever an instanceOf defined by OGML ends in an OGMLX construct, we say that it is on the OGMLX language axis.

5.5 Conclusions

Our example focused on associations in UML. We have presented a total of four ways to express UML in OGML. For three of them we were able to use the same ontological commitment. To support n-ary associations we were forced to change the ontological commitment. Full OCL support for n-ary associations was realized as some researchers consider appropriate [78].

For n-ary associations we had to define the Association as a SD. It is interesting to note here that this is ontologically correct as opposed to an association defined as MD [87]. Especially if the association also has attributes, because one ontological rule of Wand et al. forbids "properties of properties".

Another subtle difference arises with the examples about the exact correlation between inheritance and instantiation. In OGML, it becomes explicit that AssociationClass is instantiated to an Object while the Association, its generalization, is instantiated to a Link. The instantiation semantics are inherited here (the CIs and AFs are also inherited). This seems not to cause any problems but we also did not investigate it thoroughly.

With an example OCL query, we show how OGML provides the full semantics to do advanced navigation of models. Furthermore, we showed that an *ontological view*, a *linguistic view* and a *structural view* is provided for the models.

Chapter 6 – Formalization and Semantics

6.1 Introduction

In Chapter 4 we introduced OGML and proven it maps completely on OGMLX. The proof relied on some of the semantics. These only have been partially explained in Chapter 4. A complete semantics description requires a description of a *semantic domain* and a *mapping* from the *abstract syntax* to the domain. In the current chapter, we will provide the semantic domain as a graph and map the OGML constructs to it. The formalization is not complete. Detailed operations like multiplicity checking, OCL query execution and result handling are given in natural language.

For the semantic domain, we do not use first-order logic as in Section 4.6, but a graph structure. The use of first-order logic would have resulted in an axiomatic description, which is not truthful to OGMLs definition. The Clabject nature of constructs would be hard to describe in an axiomatic system. At the end of the current chapter, we will still use the graph-based domain to deduce first-order logic premises used in Section 4.6.

6.2 Semantic Domain

The semantics of the OGML language is given as an interpretation function I that maps the abstract syntax concepts defined by the OGML metamodel to the semantic domain. The semantic domain of OGML is defined as a graph as follows:

Definition 1 (Graph). *A graph G is defined as the following tuple G = (V, E, L, l, i), where*

- *$V = N \cup Lit$ is a set of vertices, being the union of a set of unlabeled nodes N and Lit is a set of labeled nodes known as literals,*
- *Labels are strings and Literals are not shared,*
- *$E \subseteq N \times L \times V$ is a set of directed labeled edges,*
- *L is a set of strings that can be used as labels,*
- *i is a function that maps edges to nodes. $i : E \rightarrow N$. i can be interpreted as a function that states from which node an edge is instantiated. We use this function in order to avoid using edges between nodes and edges.*

Abbreviated Syntax

To simplify the notation we will use the following syntax:

- *lit.value gives the label of the literal node lit,*
- *n.label returns a vertex v for which (n, l, v) $\in$ E and l = label,*
- *n.labels* = $\{l | \exists v \in V : (n, l, v) \in E\}$.

6.3 Interpretation Function

The interpretation function "*I*" provides the mapping for OGML to the abstract syntax. First, the OGMLX constructs are mapped to the *semantic domain* represented by G. Subsequently every syntactical category of the OGML abstract syntax (Section 4.3) is also mapped to it.

For OGMLX

Literal

I(Lt : Literal) = {lt | lt ∈ Lit and lt.value = Lt.value}

Furthermore there exist *dt ∈ I(OGMLDataType)* such that *lt.instanceOfOGML = dt*

Object

I(O : Object) = {o | o ∈ N}

Furthermore there exist *od ∈ (I(ode : ObjectDefinition ∪ PropertyDefinition))* and *o.instanceOfOGML = od*

Property

I(P : Property) = {p | p ∈ E}

Furthermore *l(p) = P.name* and *there exist nodes m and md* such that *i(p) = m* and *m.instanceOfOGML = md and md ∈ {Attribute, CharacterizationRelation, GeneralizationRelation*}

InstanceOfProperty

I(P : InstanceOfProperty) = {p | p ∈ E}

Furthermore *l(p) = "instanceOf" + P.language* and there exist node id such that *i(p) = id* and *id ∈ I(InstanceOfDefinition)*

SubstantialUniversal

I(SU : SubstantialUniversal) = {n | n ∈ N}

Furthermore there exist a node ***sud*** such that *n.instanceOfOGML = sud* and *sud ∈ I(SubstantialDefinition)*

MomentUniversal

I(MU : MomentUniversal) = {n | n ∈ N}

Furthermore there exist a node ***mud*** such that *n.instanceOfOGML = mud* and *mud ∈ I(MomentDefinition)*

MetaModel

I(MM : MetaModel) = {n | n ∈ N}

Furthermore OGML is a given node such that *OGML ∈ I(MetaModel)* and *n.instanceOfOGML = OGML*. Obviously *OGML.instanceOfOGML = OGML*.

Model

I(M : Model) = {n | n ∈ N}

There exist at least one label *lb ∈ n.labels* such that lb = "instanceOfOGML" and *n.lb ∈ I(MetaModel)* and *i(lb) ∈ I(InstanceOfDefinition)*.

Lemma 1: *I(MetaModel)* $\subset$ *I(Model)*. The proof is trivial and reflects the fact that MetaModel specializes Model.

For OGML

We assume that the following nodes are members of N: *Definition*, *Attribute*, *CharacterizationRelation*, *GeneralizationRelation, InstanceOfDefinition*

I(Definition) = *I*(*IndividualDefinition)* ∪ *I*(*UniversalDefinition)* ∪ *I(OGMLDataType)*

I(IndividualDefinition) = *I(ObjectDefinition)* ∪ *I(PropertyDefinition)*

I(UniversalDefinition) = *I(SubstantialDefinition)* ∪ *I(MomentDefinition)*

SubstantialDefinition

I(SD : SubstantialDefinition) = {*n* | *n* ∈ *N* ∧ *n.instanceOfOGML* = SubstantialDefinition}, *where* SubstantialDefinition *is a node in N*

MomentDefinition

I(MD : MomentDefinition) = {*n* | *n* ∈ *N* ∧ *n.instanceOfOGML* = MomentDefinition }, *where* MomentDefinition *is a node in N*

ObjectDefinition

I(D : ObjectDefinition) = {*n* | *n* ∈ *N* ∧ *n.instanceOfOGML* = ObjectDefinition }, *where* ObjectDefinition *is a node in N*

PropertyDefinition

I(D : PropertyDefinition) = {*n* | *n* ∈ *N* ∧ *n.instanceOfOGML* = PropertyDefinition }, *where* PropertyDefinition *is a node in N*

OGMLDataType

I(D : OGMLDataType) = {*n* | *n* ∈ *N* ∧ *n.instanceOfOGML* = OGMLDataType }, *where* OGMLDataType *is a node in N*

GeneralizationRelation

The inherited properties ***sps*** of superconcepts for a concept ***x*** according to language ***L*** are defined as follows:

gd ∈ *x.instanceOfOGML.container* | *gd.instanceOfOGML* = *GeneralizationDefinition* ∧
gd.instanceOfOGML ∈ *L.contens*

Each *x* can be part of the set of superconcepts *supers*:

x ∈ *supers* or

x ∈ *supers* iff exists *y* ∈ *supers* and *x* ∈ *y.(gd.generalConceptRole)* and *y.instanceOfOGML*= *gd.generalConcept*

sps is the set of all properties *p* of *supers* which have *p.instanceOfOGML* ∈ *MM.contents*

We define $\leq$ as follows: $x \leq y$ *iff* $x.instanceOfOGML = L \wedge y.instanceOfOGML = L \wedge x \in supers\ of\ y$

We define $\geq$ as follows: $x \geq y$ *iff* $y \leq x$

InstanceOfDefinition

Let *M1* and *M2* be models and *M1.instanceOfL* = *M2* where *L* is a language. Let *MM* be the metamodel that defines *L*.

Let *O* = *M1.contents*. For each $o \in O$ the following holds:

- Let *o.instanceOfL* = *d*. *d* is not empty and $d \subset M2.contents$ and the size of *d* satisfies the multiplicity defined in the InstanceOf relation for *L*.
- Let $o.instanceOfL = \{d \mid d \in M2.contents \wedge d.instanceOfOGMLX <= InstantiatableElement\}$. For each *d* we define the set of its features F in the following way. Let *d.instanceOfOGML* = *dd*, where $dd \in MM.contents$. Let *FD* is the set *{fd1, …, fdk}* where *fdi* are those labels of *dd* such that $dd.fdi \in I(CharacterizationRelation)$. *FD* also includes the labels inherited from the superconcepts of *dd* according to the OGML inheritance semantics.
- Then $F = dd.fd_1 \cup \ldots \cup dd.fd_k$.
- The set of properties *P* of *o* is defined in the following way. Let *o.instanceOfOGML* = *od*. Let *od.properties* = *{p1, …, pm}*. This set also includes the inherited properties according to OGML. The set of property labels of *o* is *PL* = *{l1 = p1.role, …, lm = pm.role}*. Then *P* = *o.l1* $\cup \ldots \cup$ *o.lm*. *o.x* can be a set.

  For a given *d*, its features *F* and for each $fl \in F$ we have the following. For *each* $f \in d.(fl.momentDefinitionRole)$ (Note: elements *f* include those inherited according to the semantics of the language L) there is exactly one property *p* of *o* such that:

  - *iod* is the substantial *IOD* of *L* found in *MM* with: *iod.definition* $\geq$ *d*d and *od* $\geq$ *iod.conformingDefinition.*
  - Let *df* = *f.instanceOfOGML* and *dp* = *p.instanceOfOGML* and *miod* is the Moment IOD of *L* found in *MM* with: *miod.definition* >= *dp* and *df* >= *miod.conformingDefinition.*
  - The corresponding CI exists. There is an $x \in iod.characterizationInstantiations\ and\ x.characterizationRole = df.momentDefinitionRole \wedge x.momentRole = dp.role$
  - There exists an AF *af* such that: $af \in miod.atytributeFunctions \wedge af.characterizationRole = df.momentDefinitionRole$
  - Both instantiation conditions evalute to true. *moir.condition* = *iod.condition* = *true*
  - The property has is an instance of *miod*: *p.(af.name)* = *af.naming* or *p.instanceOfOGML* = *miod*. The two possibilies are available because of the difference needs we defined in subsection 4.3.3. The first option supports model conformance checking and instantiation and the second supports (optimized) model querying.
  - The value of the property should conform to the typing and multiplicites dictated by *af*'s expressions: $af.lower \leq \#\{p.value\} \leq af.upper \wedge p.value.instanceOfOGML \leq af.typing$.

The CharacterizationInstantiation and Attribute functions are not formally defined. Their semantics are given in Section 4.3 in natural language.

6.4 Use of the Semantics

This semantics of instanceOf assumes that a model element is instance of its defining model element if and only if all the features of the defining element are instantiated to a property. It is possible to have properties without defining features. Still the model element will be an instance of the defining element and the processing done from the point of view of the defining element will be valid since all the features are present.

The aforementioned constraint may be strengthened by requiring that all the properties are instantiated from the features of the defining element. Perhaps both forms of the semantic checking should be implemented. One can be used to establish a *strict instanceOf relation* between models and the other to establish a *kindOf relation*. A model that is a kindOf another, can also be kindOf or strict instanceOf others.

Instantiating a Model

Each model construct has a linguistic and an ontological instanceOf. This is true for each model in OGML as was shown by Figure 4-20. One of these instanceOf relations needs to be established by the tools used for creating models; the model input facility that a modeling architecture provides. Usually this will be the linguistic instanceOf relation. However, this could just as well be the ontological instanceOf relation in, for example, the target model of a model transformation. The semantics support both the derivation of the linguistic and the ontological instanceOf relation for each construct X. It does this by assuming *X.instanceOfOGML and X.instanceOfL* and proscribing the relations that should hold under these conditions.

Supporting the OGML Bootstrap

To bootstrap of OGML is a case where the linguistic instanceOf has to be established. This was shown by the dashed arrows in

Figure 4-19 - A conceptual graph of the model constructs mapped to OGMLX

. To proof that all modeling constructs in OGML are instanceOf OGMLX, we used the following premise in first-order logic (FOL):

Premise 3: $\forall x \forall y(Ix \wedge Ey \wedge IODxy \rightarrow \forall x' \forall y'(IOy'x' \wedge IOx'x \rightarrow IOy'y))$

We define the following equivalence relations between our graph formalization and the FOL formalization:

$IOxy \Leftrightarrow x.instanceOfOGML = y$
$Ey \wedge IOxy \Leftrightarrow x.instanceOfOGMLX = y$
$Iy \wedge IOxy \Leftrightarrow x.instanceOfOGML = y$
$IODxy \Leftrightarrow x \ \ iod\ y$

Now assuming:

L = OGML	(in the graph formalism)
$\forall x \mid Ox \rightarrow Ix \vee Ex$	(first basis shown in Section 4.5 and 4.6)

We can find a complete one-to-one mapping from the variables in Premise 3 to the variables of the graph formalization (will not expand the details here). This shows the correctness of Premise 3.

6.5 Conclusions

By mapping OGML constructs to a formal structure, we expressed its semantics. The use of a graph formalization is appropriate considered the self-reflective nature of OGML. The definition of the graph structure can be almost directly translated to an implementation of a model management architecture that supports modeling and metamodeling with OGML.

We showed how first-order logic can be derived from the formalization presented here. This supports the premises used in Section 4.6.

Chapter 7 – Tool Support

In the current chapter, we investigate tool support for the OGML metalanguage. A prototype of OGML has been created to conduct the case studies of Chapter 5. Some of the insights in building this prototype are described here. Other design and implementation decisions come from literature.

7.1 Introduction

In its current version, OGML contains some high level modeling concepts as first-class entities. In OGML, the instanceOf construct defines how model constructs can be *instantiated* from an intensional model. The inheritance construct defines which kind of inheritance relations are allowed (a meta-language could for example allow only single inheritance as opposed to multiple inheritance) and what the effects of inheritance are on the instanceOf relation. For example, different meta-languages specify different semantics on this point; UML even specifies a semantic variation point allowing static features to be both inherited and not inherited.

OGML's status is best seen as a proof of concept. Its correctness and usefulness has not been proven in practice and can only be proven by case studies. To conduct these case studies we first need a tool in which the OGML concepts can be used. In the following sections, we outline a design for an OGML tool. We do this by using the standard software design process of composing requirements and creating a design. The following chapters will guide the reader through this process.

7.2 Requirements

OGML is not only a metalanguage but also a modeling architecture. Therefore, it deals with both modeling languages and models. OGML itself can be seen as a special language in which the other languages are expressed. This is expressed in Figure 7-1, where the arrows represent instanceOf relations.

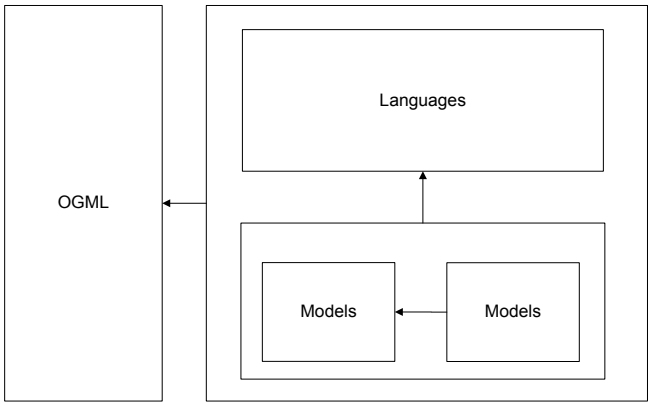


Figure 7-1 - The nested modeling architecture of OGML

The creation of a (meta)modeling environment is a great deal about supporting the pragmatics of modeling. The OGML tool thus needs to handle modeling language

definitions and models conforming to those definitions. Literature provides the following requirements for such architectures [9]:

1 – The tool should be able to do language definition
2 – The tool should be able to save and load models
4 – The tool should be able to define meta-models
3 –The tool should be able to do conformance checking of models according to their meta-model
5 – The tool should be able to import/export from/to ECore and KM3

For the MISTRAL project [65] OGML should also fulfill some requirements:

6 – The tool should be able to import languages from ECore. The imported language models do not need to use OGML's full expressivity, but just the basics, so that they can be used in MISTRAL transformations (not multilevel transformations).

Non-functional requirements include:

7 – The tool should be easy to implement and maintain, because we have limited time

Conformance Checking

Conformance checking involves the verification of the instanceOf relation between two models according to the instanceOf definition of the language. The instanceOf relation checked can be either ontological, as it is between model and metamodel, or linguistic, as It is between language and model or metamodel. Figure 7-1 illustrates this also. If we consider language definitions as models in the OGML language, than the nature of checking algorithm is the same for both ontological and linguistic instanceOf relations as we have seen in section 4.7.

A general conformance-checking algorithm would allow us to give any two OGML models (language model or normal model) as input and check whether one conforms to the other. In case of checking the ontological instanceOf relation between two models, an extra input with the language definition is needed. The result of checking should be either a set of errors referring to non-conforming model elements or a model with all the calculated instanceOf relations.

7.3 Detailed Design

The most crucial part of the OGML tool will be the conformance checking. It places constraints on the definition of OGML. Therefore, we will continuously refer to its properties to make design decisions in the next sections.

7.3.1 Modeling Space

The proof of concept tool uses the ECore reflective API for conformance checking, a downside of this is that ECore provides no explicit facilities to store instanceOf relations. In ECore models, we cannot express inter-model structures like the instanceOf relation, since it would break the metamodel conformance. Therefore, to support instanceOf relations, we need to replace ECore with a model that can represent the complete OGML modeling space [1]. Such a model should contain enough elements to incorporate any modeling architecture

in it, including OGMLs. Figure 7-2 shows the modeling space and its place in the OGML architecture.

With the first class entity for the instanceOf relation in the modeling space, we can explicitly represent both the linguistic and the ontological relations between models and their elements. Since OGML language definitions define the semantics of these instanceOf relations, it needs to be able to refer to the elements in the modeling space. OGML contains an ontological representation of the modeling space just for this purpose. This is displayed in Figure 7-2 with the line with dotted ends.

The modeling space can be implemented on any modeling architecture of choice. We can even opt for a proprietary implementation in any programming language. However to reuse existing modeling tools (and fulfill requirement 6) we opt for ECore to implement the modeling architecture in. This will not allow us to define the modeling space into itself, as would a proprietary implementation. In addition, it will make the picture not symmetric, because OGML is expressed in itself. However, these things do not limit our ability to do conformance checking, since the elements of the modeling space may always be looked up by name in OGML.

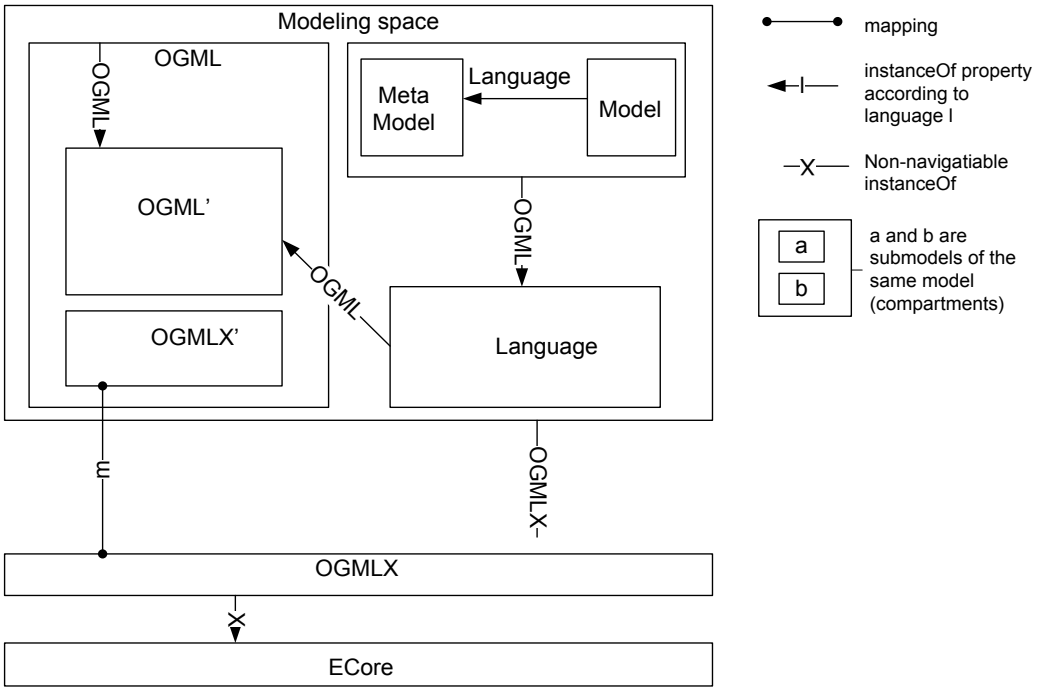


Figure 7-2 - The place of the modeling space in the OGML architecture

In Section 3.1 we have seen that conformance checking can be done on any model in the OGML architecture, the checking algorithm is independent of the sort of models we may choose: language models, normal models or the OGML models itself. Thus, in order to be able to implement one generic checking algorithm, we need one API for all the models. Therefore, we will also need facilities to incorporate all models into the OGML modeling space.

7.3.2 Handling Languages

Since we opt for the use of KM3 and TCS, languages can be written by a user in text. From this text, TCS extracts an ECore model conforming to the OGML abstract syntax metamodel as shown in Chapter 4. These models will have to be transformed to the OGML modeling space. A transformation should preserve all linguistic instanceOf relations of the language models as some first-class entity in the modeling space. These all refer to the OGML language model, therefore the OGML language should first be transformed to the modeling space. The transformation is shown in Figure 7-3 as T(o2ms). The reflective API of ECore can be used to handle all elements in the OGML metamodel uniformly where needed.

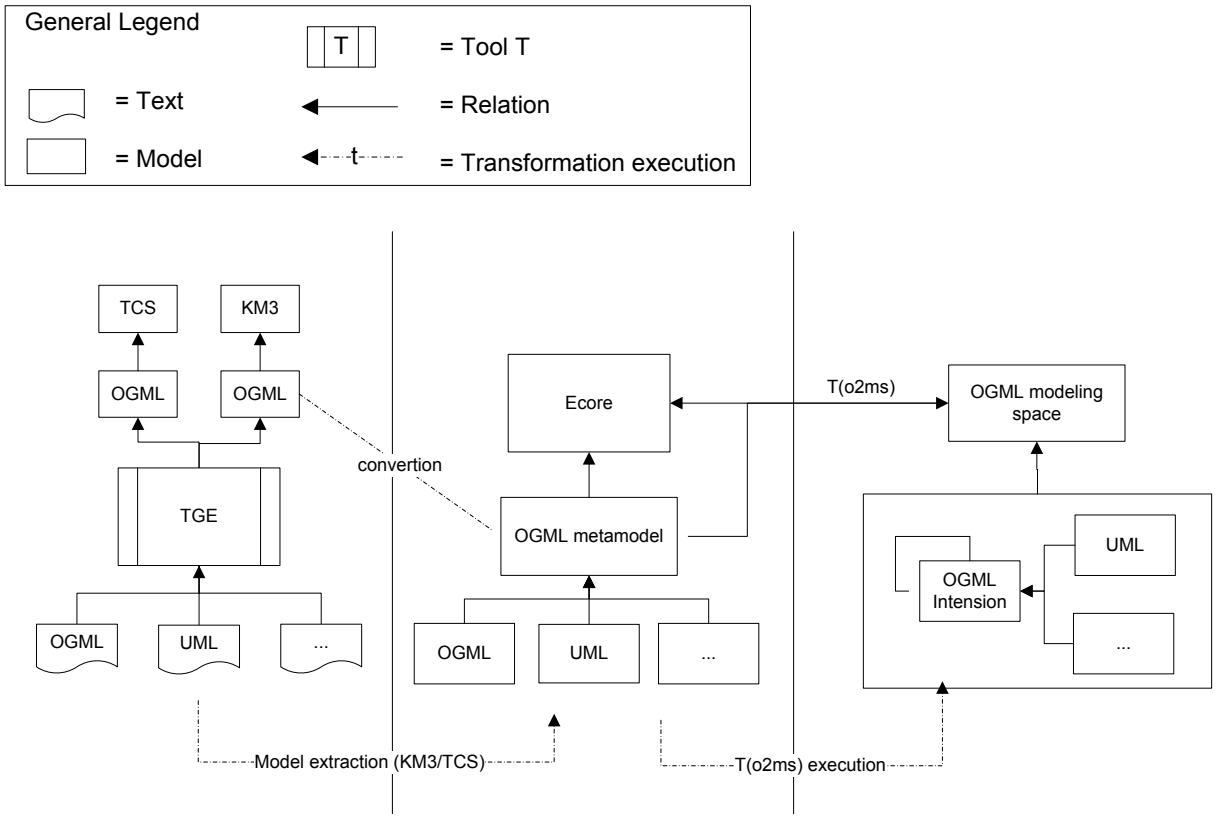


Figure 7-3 - OGML with modeling space

In the section about MISTRAL, we also consider a way to import languages in an incomplete but sufficing manner.

7.3.3 Handling Models

For every language defined in OGML, a user would want to create models in it. We can identify two use-cases here:

1 – The language already exists with appropriate tooling. In this case, the user would want to use this specific tooling to define his models, since OGMLs generality will hardly allow to create better tooling on top of it.

2 –The language does not exist yet or no appropriate tooling is at hand. In this case, the user needs some basic facilities to define a model.

To solve the first use-case we have to provide a transformation from the model architecture of the tool to the modeling space. This is illustrated in Figure 7-4. The two UML models should not be confused in this figure, one represents the UML metamodel as used by the tool and the other is the UML language model in OGML as specified by the user and transformed to the modeling space. The transformation involves resolving the linguistic defining type of each model element in the language definition and creating instanceOf relations for them. Therefore, the language definition should be an input of the transformation.

In Figure 7-4, we also see that the UML tool uses ECore as modeling architecture. By using the reflective capabilities of ECore, we can spare us the effort to write a rule for every individual UML construct. Instead, we just refer to the ECore elements and map these by name to the elements in the language model. This lifts the level of the transformation, making it effectively T(ecore2ms). A requirement for this approach would be that the names of the UML definition in OGML are the same as those used in the UML metamodel of the tool. Any modeling architecture other than ECore can be supported with a separate transformation; as long as it provides the same reflective capabilities as ECore, (RDF meets this requirement).

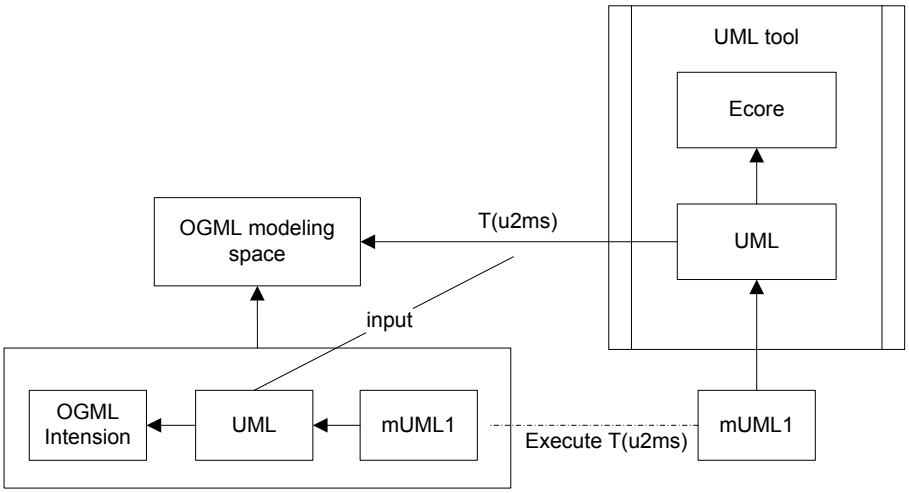


Figure 7-4 - Importing models into the modeling space

For use case two, we need a concrete syntax for writing models. OGML has no means to express the concrete syntax of a language, only the abstract syntax structure and (part of) the semantics can be defined. A user of OGML will have to write his own syntax parser for every language for which he wants to create models. To nonetheless be able to provide a way to write models for OGML, we decide to create a concrete syntax for the modeling space (see Figure 7-5). We realize that such a concrete syntax provides however no direct feedback on the linguistic conformance of the model being written. In a later stage, we may choose to generate standard concrete syntaxes parameterized by the language abstract syntax as is illustrated in Figure 7-6. The transformation T(ecore2ms) can be reused here.

The output of an implementation can be checked for linguistic conformance to OGML by the conformance checker.

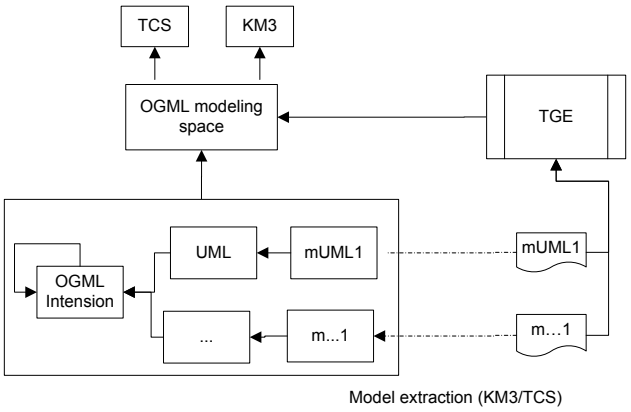


Figure 7-5 - A concrete syntax for the modeling space

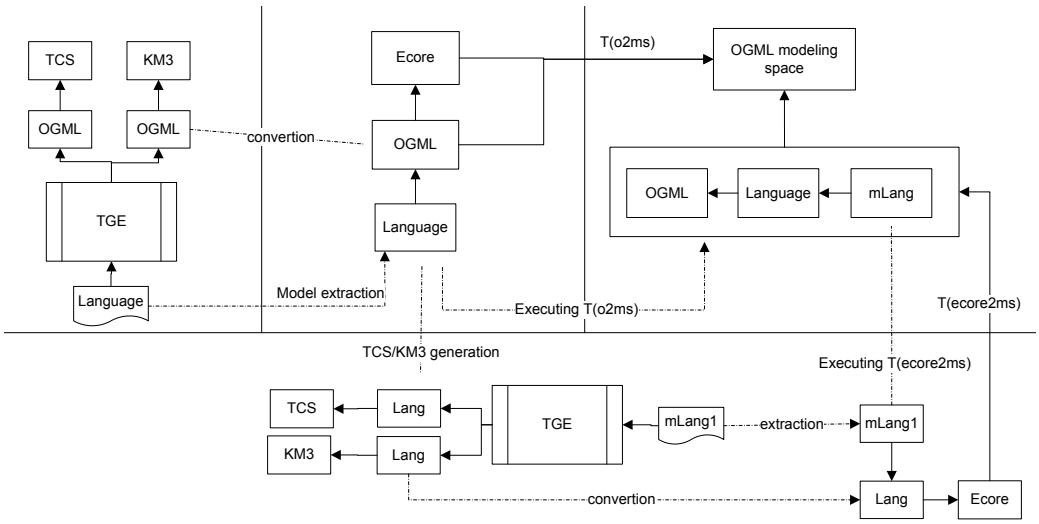


Figure 7-6 - Parameterized syntax generation

7.3.4 OCL

The conformance checking of instanceOf relations involves some model querying. For example in the presence of multiplicity, we need to check how many model elements are there, or for a typed language we need to look up the type for each attribute and see if the assigned element indeed conforms to this type. For these model queries OGML makes use of OCL. However, since our models are expressed in the modeling space, we cannot reuse any existing OCL implementations. An OCL interpreter needs to be implemented that can take OCL programs expressed in the modeling space as input. These OCL programs are incorporated in the language definition; therefore, the interpreter needs an extra input from the environment where OCL variables can be bound to elements in the language definition.

OCL semantics [73] always assume linguistic instanceOf relations. Since OGML threats the linguistic and ontological instanceOf relation in the same way, the normal semantics of OCL

do not suffice. In addition, the semantics for any construct in OCL that refers to model elements should be refined.

7.3.5 Conformance Checking

With the design choice for the modeling space containing all models for language definitions, models and OGML itself, conformance checking becomes a straightforward task. The conformance checker should take two models as input. In case of checking the ontological instanceOf relation between two models, an extra input with the language definition is needed. The OGML language definition is always loaded, because the language definitions refer to it.

The checker makes use of the fact that any model has at least two instanceOf relations; one linguistic and one ontological. Since these two relations are known for OGML (it is instance of itself and of the modeling space), they can be derived for any (language) model in the modeling space. Because all models directly or indirectly are instanceOf OGML and the linguistic instanceOf for all models has been recorded as described in sections 4.3 and 4.4.

Since we choose implement the modeling space in ECore, the checker can be written in any language that can easily handle ECore input. The OCL implementation can be reused for model querying.

7.3.6 A MISTRAL Use- Case

A goal of MISTRAL is to do model architecture independent transformations. MISTRAL can do this by using explicit instanceOf relations making OGML a perfect candidate to provide its input and output models. Key scenarios supported by MISTRAL are transformations between models in different model architectures and transformation executions over multiple model levels. However, other scenarios included are incremental updates and control over the execution order. Since the latter scenarios are not present in any other transformation language, a user may choose to use MISTRAL specifically for this and not use the model architecture independence.

An implementation of MISTRAL that uses OGML takes only OGML inputs. Therefore, a user would always have to go through the tedious effort of writing a language specification in OGML, which will only be partly used by the MISTRAL engine. An automatic import of languages is however not possible, since OGML has other constructs than the language being transformed. We can however create a flat import of the language, mapping the language on basic OGML constructs.

7.4 Architectural Design

Figure 7-7 shows the OGML architecture. It consists out of a repository of models in the OGML modeling space, transformations to update this modeling space and programs which work on the modeling space. In the figure, we abstracted from document parsing, by leaving out the TCS parsers. All metamodels are expressed in ECore.

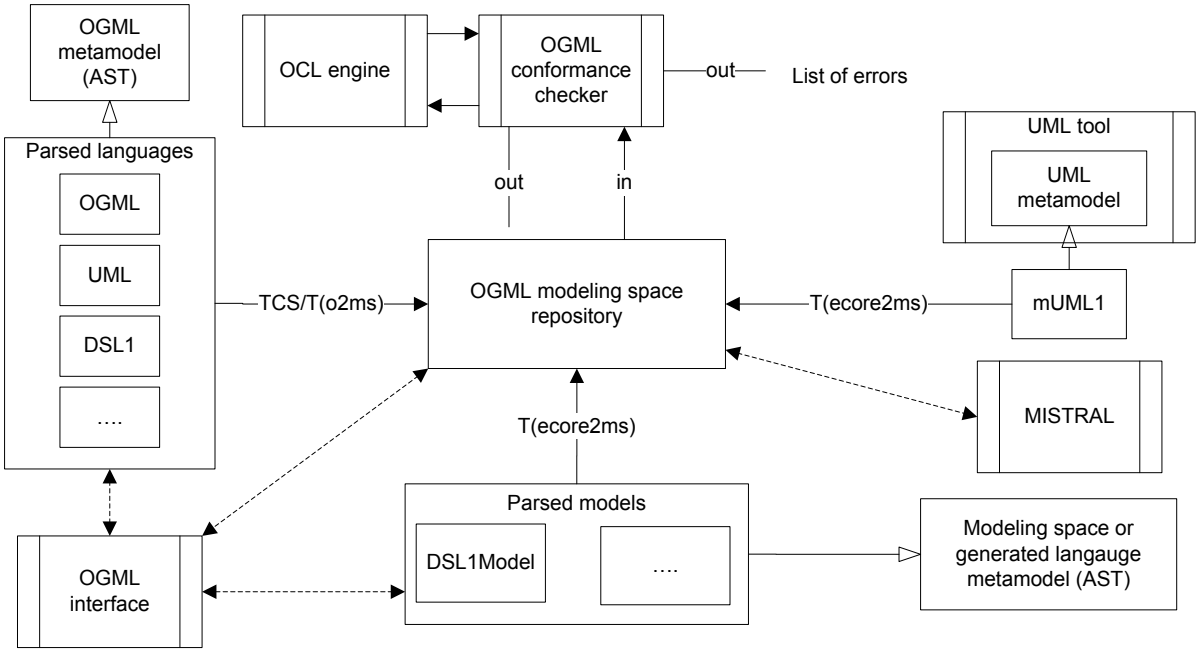


Figure 7-7 - OGML architecture

The interface design will be discussed the next section.

7.5 Interface Design

The design of the OGML tool includes models and transformations between them. For every OGML use-case, we witness the creation of more models in and around the OGML modeling space. Providing the OGML tools separately will most probably result in erroneous usage and inconsistent modeling hierarchies. Moreover, even if a user succeeds in finding an appropriate means to work with the tools, he will experience difficulties sharing his work with other users. For these reasons, we realize an automated handling of the tools is needed. In the current chapter, we will provide a possible organization for an OGML perspective in Eclipse. Eclipse is chosen for its ability to handle ECore models and automate related tasks.

The perspective

The OGML perspective can provide a view on OGML projects. Each OGML project consists of a language folder and a models folder.

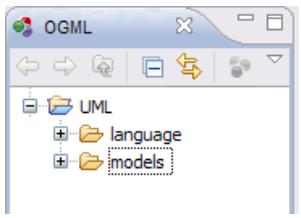


Figure 7-8 - OGML perspective screen

Use-case language creation

Languages can be specified as text. TGE provides direct feedback on linguistic conformance to OGML. The source file will be automatically compiled and put into the modeling space when the file is saved. Just like the default Eclipse behavior.

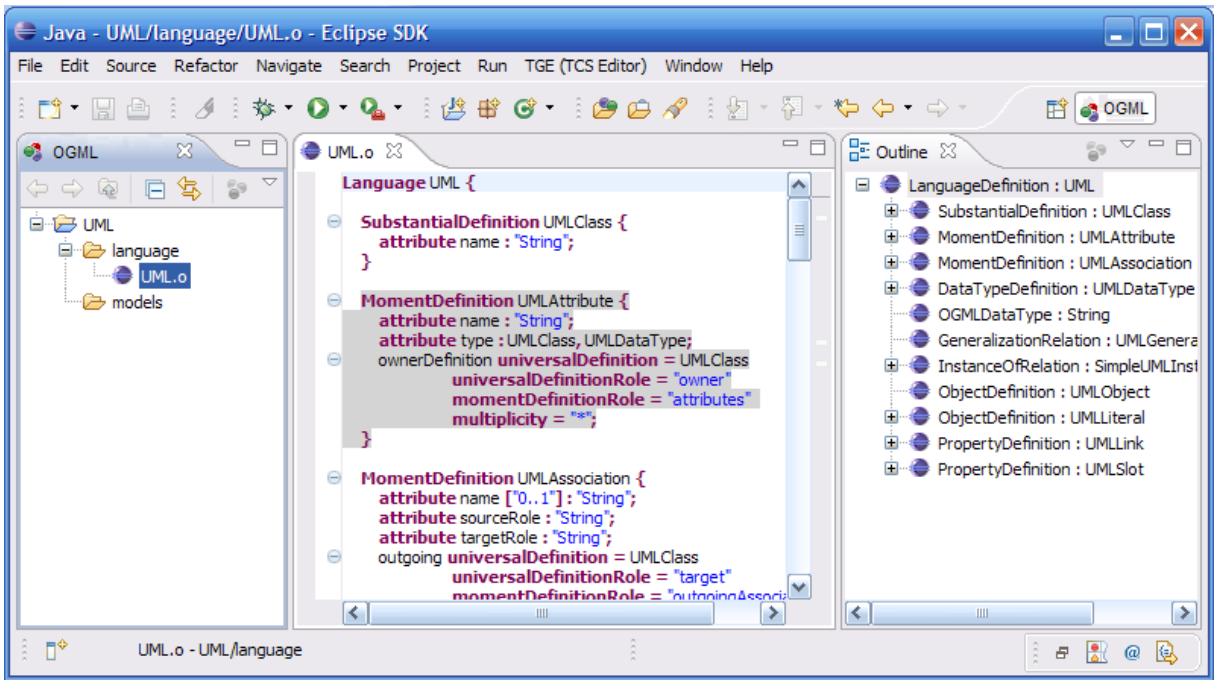


Figure 7-9 - Language creation screen

Use-case model import

A language has to be supplied as argument, either in a separate question box, or by selection in the language definition in the OGML Explorer as is shown in Figure 7-10.

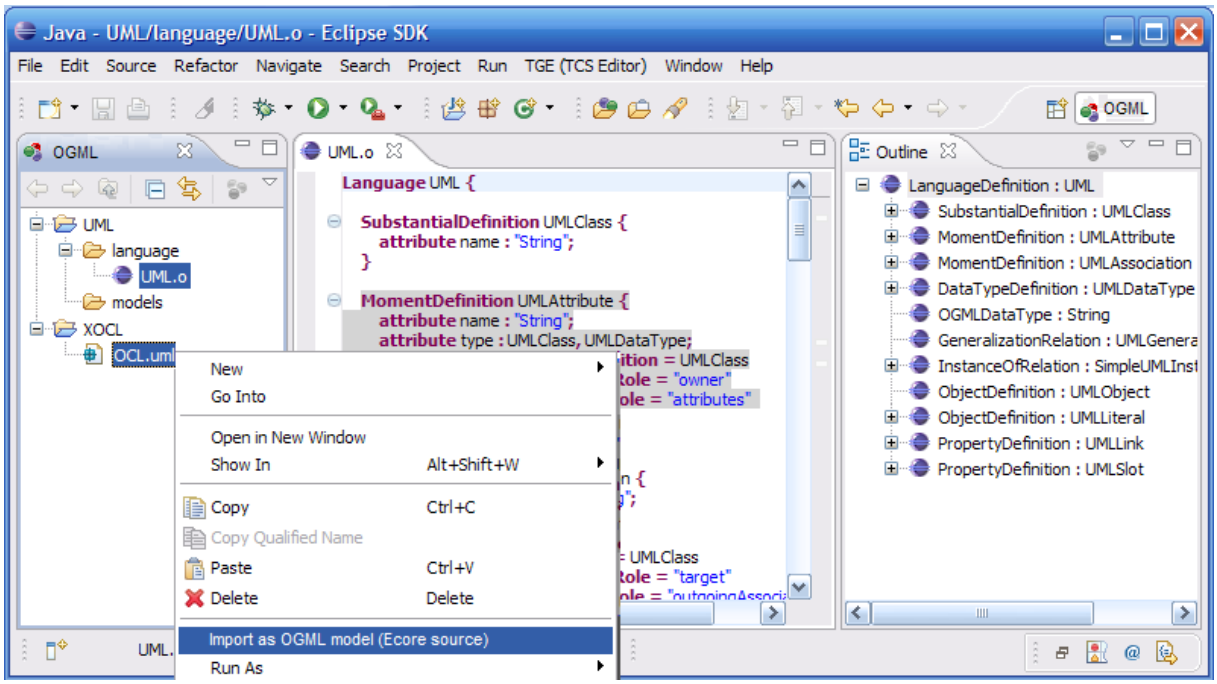


Figure 7-10 - Import model screen

Use-case model creation

Like language creating, we can create a file with the appropriate checking and start editing. For future designs containing generated standard syntaxes, we may have to provide an extra

facility to generate the syntax and to bind it to a specific file extension. Any loaded models will be shown in the explorer.

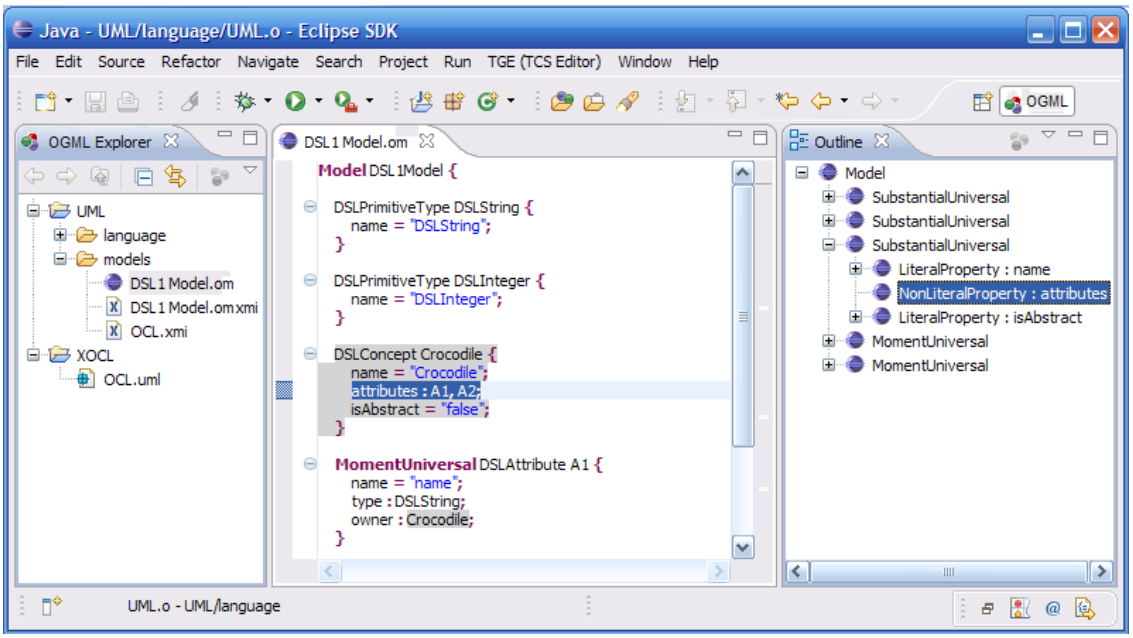


Figure 7-11 - Model creation screen

Use case conformance checking

Any set of models and languages can be selected in the OGML Explorer. When the OGML conformance check is invoked and it asks which language axe to check.

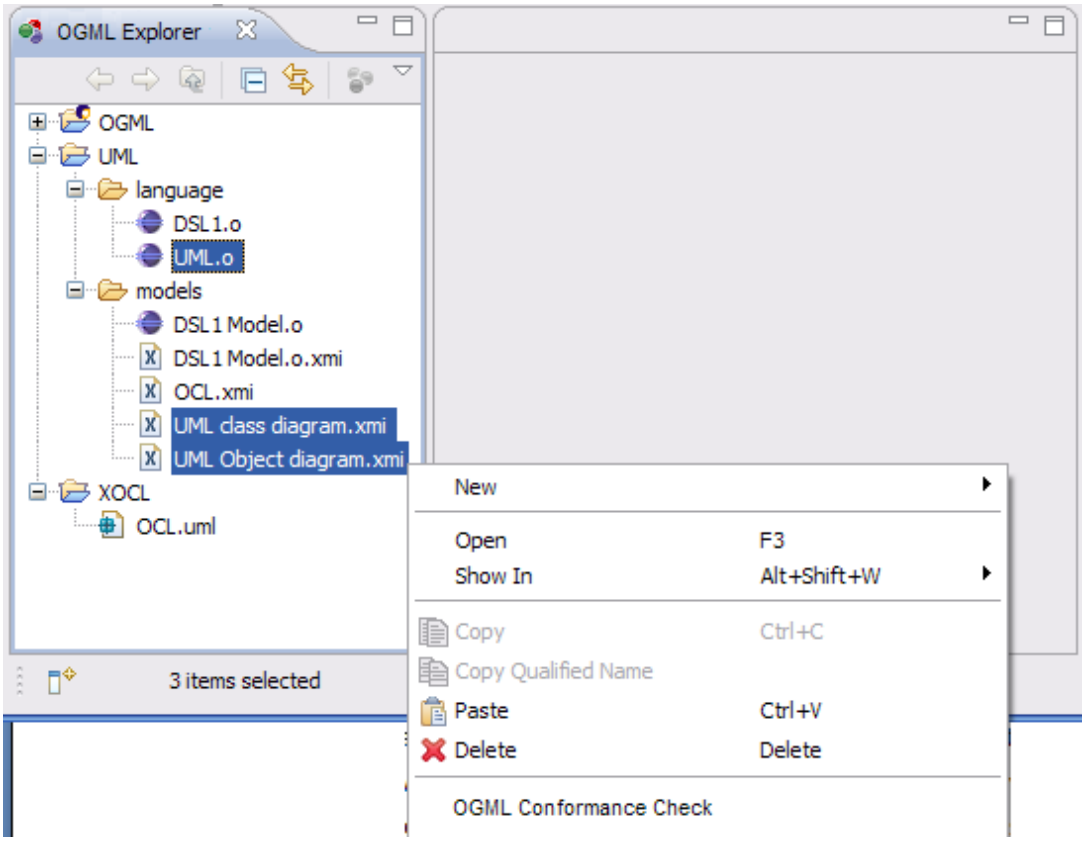


Figure 7-12 - OGML conformance check screen

7.6 Extensions

Modeling Space

A modeling space should allow all different modeling architectures to be incorporated in it (See Chapter 4). UML uses the name property in meta-models to identify modeling elements. <another example>. Since a truly modeling architecture independent structure could not decide on a specific identity for modeling elements, we have to provide

OGML

For certain model query operations, it is necessary to retrieve the multiplicity of a model. This multiplicity depends on the language of the model. Therefore, to retrieve this multiplicity OGML needs to represent this multiplicity explicitly, which is currently not present in the language.

Current Generalization-specialization in OGML is expressed in static manner; a language can only specify specialization, but not the precise effects of specialization. To express this, general constraints on language elements can be used in conjunction with a natural deduction language. Like OCL, the implementation of the latter should operate on the modeling space elements.

7.7 Conclusions

We have seen that OGML can be implemented on top of an existing modeling architecture, thereby saving time in the development process. The expected benefit of OGMLX was shown; we mapped it directly to ECore in the prototype implementation. All models could be handled uniformly.

The prototype relies extensively on EMF (for model storage), ATL model transformations (for uniform representation) and TCS (for syntax). These technologies are used for model input, output and storage. OCL has been implemented to support the OGML `InstanceOfDefinitions`. OCL has been reused independently to support querying on OGML models. The `LanguageAxeExpression` allows querying according to the language of a model, according to OGML (intensional) and according to OGMLX (OGML extensional). Via the OGML instanceOf definitions the queries cascade from language (of the model) to OGML and finally to OGMLX.

To ensure correct models a model conformance checker has been implemented. It support checking of correctness of models according to their language. In this manner, the checker supports modeling and metamodeling.

Chapter 8 – Related Work

8.1 Introduction

The current chapter presents related work and compares it with our approach, proposal and results. First, we look at work that preceded OGML. Second, we look at work that takes a similar approach to ours and we compare our results with it. Third, we treat some works that try to solve similar problems. We found work in the field of data engineering and another that focuses on improving OCL.

Several related works aim at providing a consistent interpretation of tradition modeling architectures using argumentation based on observations. We treat these more theoretical works in the end of the current chapter.

8.2 Earlier Work

Kurtev devoted his PhD thesis [62] to "adaptability of model transformations". He found that the limitations in transformation languages require a uniform definition of the instanceOf relation in the modeling architecture. He investigated the instanceOf relations that can be found in the MOF architecture. He represented them in Figure 8-1. The figure is detailed and we will not explain it fully here. A short explanation suffices: the layers and the instanceOf relations can be mapped one-to-one to the OGML architecture in Figure 4-20. The main idea behind OGML of making the instanceOf relations explicit stems from this work.

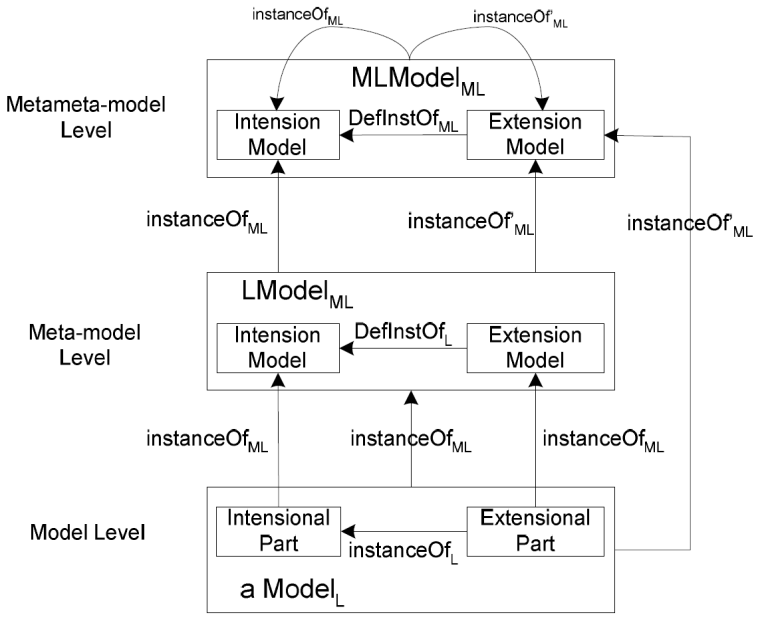


Figure 8-1 - InstanceOf relations found in MOF architecture (taken from [62])

In "Metamodels – Structural Definitions or Ontological Commitments?" [62], Kurtev presented the ideas behind OGML. The work mainly focuses on the use of Ontology in metamodeling. It features an initial proposal for a self-reflective definition of the ontological constructs in OGML. An ontological definition of the instanceOf relation and the

generalization relation is given in this paper. The metalanguages, which we proposed in this thesis, uses this previous version of OGML as a basis and extends it with an explicit definition of (the semantics of) the instanceOf relation.

8.3 Related Work in Approach

We found several works that take a similar approach to ours with to solve the problems identified in Chapter 3.

Formal Ontology

Evermann et al. [30] use the Formal Ontology for the grounding of existing modeling languages. They give an interpretation function for (UML) model constructs in terms of the BWW ontology. This allows them to analyze the ontological correctness of these constructs. These works thereby provide valuable insights in the nature and meaning of the constructs. Guarino et al. [39] and Wand et al. [87] both give a precise definition of attribution . Wand relates this to current modeling practices found in UML and MOF. Wand also proposes a set of rules, which could guide the use of constructs in modeling. Guarino's and Wand's work is also based on the BWW ontology.

Guizzardi et al. [44][45] and Degen et al. [26] propose an "upper layer ontology" based on FCO. This could be used as a metamodel. They use detailed subcategories of universals: sortals, rigid, anti-rigid, non-rigid, etc. They give mathematical descriptions from ontology to formalize the constraints on language constructs.

Although these works provide useful contributions to the modeling community, they do not aim at providing complete solutions for modeling and metamodeling. They do however tackle the problem of ***lack of real-world relation*** (relating back to Chapter 1).

Describing the Modeling Architecture

Gogolla et al. [37] give a formal representation of the traditional modeling architecture. They only considered UML and MOF. They described all models in the MOF modeling architecture using a minimal model in the form of a graph. Using OCL queries, they express the relations between the different layers in the architecture. They conclude with some interesting questions, which are related to our OGML results:

- Are the (instanceOf) relations between the models dependent on the languages? In the approach of OGML, this question is answered positively.
- What is the optimal minimal model in terms of understandability? They propose to condense it to a single node. From ontological point of view, this would indeed create more confusion, since it breaks several Rules for truthful real-world representations [87][42]. In our minimal model: OGMLX, we were able to preserve as much real-world knowledge without creating any ambiguity: the main taxonomy of OGMLX conforms to the ontological square of FCO. From the point of view of the metalanguage, the choice for an OGMLX construct is always determined precisely.
- Is it possible to have layer-dependent constraints on the model? Recognizing meta- and modeling languages as semantics descriptions for the instanceOf relations, as OGML

does, allows exactly this. In the next chapter, we discuss more potential gains from the instanceOf definition in OGML.

- What is the "typing"[16] of the `UMLAssociation` in terms of MOF? The "typing" of the `UMLAssociation` is found to map differently to `MOFAssociations` depending on their order (binary, ternary or n-ary). This result is similar to the different ways we were able to express the `UMLAssociation` in Chapter 5.

Bézivin [18] uses conceptual graphs [85] to do essentially the same thing as Gogolla: representing and investigating the modeling architecture. The use of conceptual graphs is "in order to stay as language-independent as possible". Bézivin uses a different paradigm: programming languages. This approach is taken to avoid the confusion that arises when thinking in terms of the modeling architecture.

Both attempts take the approach to use *neutral terminology* for their constructs. This has the advantage that the meaning of new constructs is not entangled with old interpretations. For example, using the word "class" could create confusion among different receivers of the message who may all relate it to "classes" in different languages: an UML "class" or a MOF "class" which is a metaclass whose instantiation semantics work at a different level in the modeling hierarchy. Like Bézivin's and Gogolla's framework, OGML is incomplete and currently does not provide a constructs for the MOF construct of `Package`.

Both works provide good conceptual insides on aspects of the traditional modeling architecture. Gogolla et al. [37] shows the existence of different instanceOf relations (that is: linguistic and ontological) in the modeling architecture. They also succeeded in formalizing some of the notions that surround more complex instantiation mechanisms as proposed by Atkinson [7]. Bézivin compares the multiple instantiation with similar properties of object-oriented languages like SmallTalk. In this way, he makes a persuasive argument for the use of inheritance at the M0 level. Atkinson and Kühne [17] propose the same.

These works also take the ***approach of investigating the modeling architecture and proposing additions***. They do however not extent the semantics of the metalanguage. In both approaches metamodeling remains at the level of giving structural definitions. Using neutral terminology also cannot provide guidelines for modeling as Formal Ontology does.

Lack of Modeling Constructs and Language Semantics

In Chapter 1, we analyzed problems with traditional modeling architectures. We found a ***lack of constructs*** to be the source of or at least contributing to some problems. Several other works recognize the same problem and try to come up with a solution. Evans et al. take the approach of recording the instanceOf relation with a structural mapping between each modeling layer [1][29].

Atkinson and Kühne provide three solutions: multi-dimensional modeling, potency and Clabject ([87], [4] and [5]). Gitzel and Hildebrand propose to introduce an explicit element for the "model layer". These works have in common that they reason from the point of view of

[16] The papers terminology for instanceOf

the existing modeling architecture. The insights presented are valuable for the understanding of the architectures

In Chapter 1, we also identified a ***lack of language semantics*** as a limitation for metamodeling. Soden [82] apparently shares this view. His approach is to add a construct for instanceOf to the metamodel and extends metamodel with its semantics definition.

However, all of the aforementioned approaches do not provide uniform construct handling across all layers (or meta level independent modeling [5]). These approaches do not solve the cross-language interoperability because they are on the level of the modeling language.

8.4 Related Work with the Same Objective

In this section, we discuss works that solve similar problems as ours. We start with example from the field of data engineering. Then we show a work that proposes an OCL Interpreter, which is language independent.

Data Technology

Atzeni, Cappellari and Bernstein [13][12] consider methods for data translation over two levels: schema and data. To this extent, they express the schemas and the data hierarchy of schemas. The hierarchy is implemented in database tables and resembles a modeling architecture. The ontology they use for the "metamodels" comes from database research [48]. "Multilevel dictionaries" are used to perform the translation.

Their work is a contribution to the problem of multilevel transformations as we discussed in Chapter 3. For us it is interesting to learn that the database world uses similar techniques to those of MDE. From the point of view of modeling, we make some observations: it does have an implicit real-world relation in database engineering. Both the technological and the conceptual side of the work are in the field of data engineering. This limits the applicability of the approach. Of course, MDE and database technologies have different goals, but *both* fields have to deal with conceptualization of data *and* physical data representation in the end.

Language Independent Model Processing

Kovolos [57] was inspired by the work of Kurtev [63] in the field of model transformations and made a language independent OCL Interpreter. The approach is an early attempt to generalize the OMG OCL specification in order to support a metamodeling environment. This is useful contribution since OMG does not provide a solution for it. Because the approach is less fundamental than ours is (they do not propose a new modeling architecture), it is comparatively less integrated with the modeling environment. Whereas Kovolos's work requires a metamodeler to specify a triple of queries for each language, OGML includes the navigation semantics in the language specification as shown in Section 4.3.

8.5 Related Work in Theory

Instantiation

Modeling has been successfully applied in other technological spaces [20][64]. Data description languages like XML and RDF are suitable candidates to model. They differ in

that they offer different ways to describe the structure of data (schemas, DTDs, etc) and the associated instanceOf relations [24].

Modeling architectures and their design

Atkinson and Kühne [6][10] report that MOF specifies that the modeling architecture is linear and **strictly** layered, meaning that it only permits instantiation between two subsequent layers. The original purpose of MOF, however, was a data repository: it provides structure for all models. The interpretation of MOF changed to a conceptual language for modeling languages, a metalanguage. With the original interpretation still intact, this *strict* metamodeling can in fact not be adhered to. Related work often inspects the details of the MOF architecture [32][34].

8.6 Conclusions

In Section 8.2, we showed previous publications on the ideas behind OGML. The latest work from Kurtev [63] contains an initial proposal for a self-reflective and ontology based metalanguage.

In Section 8.3, we showed work with approaches similar to ours. Each tried to find a solution for the problems in MDE. None of them lifts the semantics of the metalanguage from a structural definition to a description of semantics. All of them provide valuable insights in the complexities of modeling and metamodeling.

Section 8.4 discussed work with the same objective. We found a work in data engineering, where the authors propose a solution for the data translation problem presented in Section 3.5. Another work proposes an improved OCL that can deal with models in different modeling languages.

From the theoretical side we see several discussions on the nature of specific modeling semantics. Those were summarized in Section 8.5. Several approaches are provided to define a uniform definition of the instanceOf relation. Most of them, however, lack a proof and only compare their solutions with different approaches. Still they provide good insights in the different aspects that make up a modeling architecture. Some of them could be used to evaluate different design decisions for these architectures.

After investigating related work, we can conclude:

- We aim at solving relevant problems. All the "related work in theory" tries to provide a solution for uniform model handling (or "metalevel independent modeling" [5]).
- We provide a unique contribution. The approach in this thesis is to propose a metalanguage based on Ontology with explicit instantiation. This integrated approach is not chosen before. Nor did any other work aim at dealing with the problems on the level of the metalanguage.

All of these works contributed greatly to our understanding of the problems that we dealing with.

Chapter 9 – Conclusion

9.1 Introduction

In this thesis, we proposed an approach to metamodeling with the goal to solve an identified set of existing problems in the field of MDE. The result is a metalanguage, called OGML, which is supported by modeling tools. In the current chapter, we give the overall conclusions for the thesis. First, in Section 9.2, we summarize this work in terms of the solution domain. In Section 9.3, we evaluate the results in the light of the initially formulated research questions. The section also reflects on the extent to which the research objectives have been met. In Section 9.4, we discuss these results and their possible application. Possible extensions and future developments are outlined in Section 9.5.

9.2 Summary

In the following subsubsections, we summarize the background, the problem description, the approach and the results we achieved.

Concepts: MDE, Languages and Ontology

In Chapter 2 "Background", the concepts used in this thesis were explained starting with Ontology in Section 2.2. Ontology is the study of *existence* it tries to categorize the entities in the real world. In Formal Ontology relies on logic to express the relations between the categories. We further explored Ontology with an emphasis on the four-category ontology (FCO): a particular ontology that recognizes the existence of *universals*.

In Section 2.3, we explained important aspects of languages; semantics (domain) and abstract and concrete syntax. The abstract syntax of a language can be expressed in a model. Languages can be seen as *ontological commitments* [77] that thereby define their own view on the world. What is expressible in a language depends on this commitment. Language definition (including metamodeling and the creation of a metalanguage) is a matter of balancing between expressiveness and precision.

In Section 2.4, we summarized the core concepts in *modeling* and *metamodeling*. We assumed definitions for the concepts of *model* and *metamodel* using knowledge from several scientific fields. In the context of modeling, we explained the *instanceOf* relation and related terminology. *Linguistic* and *ontological* instanceOf relations were distinguished. In the context of metamodeling, we explained how these concepts have a relative meaning. Therefore, the terms *intension* and *extension* were introduced.

Problems in MDE and an Approach to Solve Them

Chapter 3 "Identification of Problems in Contemporary Modeling Architectures" identifies the problems in both modeling and metamodeling areas. Traditional modeling has the potency to be *imprecise* and *inconsistent* because of a *lack of real-world relation*. For metamodeling, the matters are worse in our opinion. Problems in modeling architectures were summed up in sections 3.3 to 3.5. They seem to be related with the *instanceOf relation* and *meta-layer independent handling* of its semantics. This all results in limited possibilities for *automation* and *model reuse* in MDE.

We identified a solution domain based on our problem analysis. The analysis revealed three main problems: *a lack of real-world relation*, *a lack of modeling constructs* and *a lack of language semantics*. In order to solve the problems we took an integrated approach that uses knowledge from the domain of Ontology and extends the metamodeling with explicit constructs and semantics for *instantiation*. Ontology could provide us with clear guidelines for modeling and metamodeling practices and a more explicit notion of instantiation could lift the practice of metamodeling from the level of structural definitions to full semantic language descriptions.

Proposed Solution

In Chapter 4 "An Ontology-Based Modeling Architecture", we presented Ontology-Grounded MetaLanguage (OGML). OGML implements the approach in the following ways:

- The constructs from this metalanguage are drawn from Ontology and especially four-category ontology; *universals*, *individuals*, *moments* and *substantials* are among the constructs that the language provides,
- It has constructs for defining instantiation semantics,
- It has constructs for explicitly representing instanceOf relations in models.

A running example was used to explain the language. A semantics description was given within this explanation.

The metamodeling practice in OGML is clearly separated from the modeling practice. Metamodeling requires specifying fine-grained ontological commitments and instantiation definitions. Modeling is an activity of capturing information about the state of affairs in a certain domain. Modeling is guided (and restricted) by the chosen ontological view expressed in the metamodel. As we showed with examples, *Formal Ontology provides the guidelines* for both practices.

The end of Chapter 4 (Section 4.4 to 4.7) was devoted to the explanation of OGML modeling architecture. We introduced *OGML eXtensional* (OGMLX), an ontology-grounded structural model. It is *part of and expressed in OGML*. Then we showed how OGML is expressed in itself. With this self-reflective definition, OGML also defines its own instantiation semantics. The semantics definitions maps OGML constructs to OGMLX constructs. In this way, every model in the modeling architecture becomes an instance of OGMLX. We have proven this using a first-order logic premises derived from the semantics definition. Thus, OGML provides *a uniform structural representation of all models*.

We concluded chapter four with observations about the modeling architecture. OGML has a *nested* modeling architecture with *three layers* and a *compaction level* formed by OGMLX. OGMLX is a *library format* for all models and languages and *also* a *language format* from the point of view of OGML[17].

[17] Terminology and concepts are explained in [11].

In Chapter 5 "Case Studies", the running example of Chapter 4 was extended with UML `Associations`. This was done using different structural definitions for the UML Object model and also by extending the ontological commitment of SimpleUML1 in SimpleUML2. This showed some of the capabilities of OGML. Furthermore, the ontological nature of UML constructs became apparent by expressing it in OGML, just as expected. With an example OCL query, we show how OGML provides the full semantics to do advanced navigation of models.

Furthermore, we showed that indeed a model in OGML can be navigated according to the view of at least two languages. The example model in Chapter 5 we navigated according to: UML, OGML and OGMLX. This "language perspective" is made explicit by the OCL Interpreter with a *LanguageAxis* expression.

In Chapter 6 "Formalization and Semantics", we give the semantics to OGML by mapping it, together with OGMLX, to a graph structure. We saw that there can be multiple uses of the semantics. This is necessary because, the *OCL Interpreter*, the *model conformance checker* and *model input tools* need different algorithms. The semantics definition allows reconstructing both the ontological *and* the linguistic instanceOf relation of models. Thereby the different algorithms can be arrived from it[18]. These algorithms are needed in the tasks of *navigation*, *conformance checking* and *instantiation* of models. Furthermore, it gives freedom to the tool design because the intensional models from both the linguistic and the ontological instanceOf can be used as abstract syntax.

The use of a graph formalization is appropriate considered the self-reflective nature of OGML. We showed how first-order logic can be derived from the graph formalization. This supports the premises of Section 4.6.

Chapter 7 "Tool Support" explains in detail the realized prototype. The prototype relies extensively on EMF (for model storage), ATL model transformations (for uniform representation) and TCS (for syntax). The prototype matches the semantics of OGML described in Chapter 4 and Chapter 6. Model input and export is not yet supported by the prototype. It is however possible to use it as a general modeling tool, because a generic syntax is made, which can express OGMLX models.

Related Work

Chapter 8 presents "Related Work". OGML is based on earlier works from Kurtev [62][63], which are presented in the beginning of Chapter 8. Then we focused on some works that used a similar approach. Several are found with varying objectives. Some try to solve the problem of real-world relation with Ontology. Others try to create uniform model handling by proposing new architectures or by studying the architecture. Although their results are useful (also for the realization of OGML), none of them provide a definition of the instanceOf semantics in the metalanguage.

[18] we did not show this but know it from the experience of implementing the prototype

We also found work with the same objective. In data engineering, Atzeni et al. propose a solution for the data translation problem presented in Section 3.5. Another work proposes an improved OCL that can deal with models in different modeling languages.

From the theoretical side we see several discussions on the nature of specific modeling semantics. Several approaches are provided to define a uniform definition of the instanceOf relation. Most of them, however, lack a proof and only compare their solutions with different approaches.

All these works provided good insights for our work. At the same time, we concluded that our work provides a unique contribution in the sense that no other work tries to solve the problems at the level of the metalanguage.

9.3 Evaluation

Here we investigate the results we achieved in the light of the research questions and the research objectives.

Evaluation of Research Questions

In Chapter 1**,** we formulated the following questions:

RQ 1*: What can we use as a solution domain for metamodeling?*

RQ 2*: How to express instantiation uniformly in a modeling architecture?*

To answer **RQ1**, we made a motivated choice for the use of Formal Ontology as a solution domain. The domain proved useable to make design choices regarding metalanguage and modeling architecture and also to support the modeling and metamodeling activities. This was expected, since other work already demonstrated the applicability of ontologies in metamodeling practices.

To answer **RQ2** we have taken the approach to enhance the metalanguage with additional constructs. In OGML, modeling languages can contain full descriptions for the instantiation semantics. The reasoning behind this goes in two steps: several problems can be witnessed in traditional architectures that involve the instanceOf relation, which has no layer-independent interpretation. This relation has different semantics specified by the language and is therefore relative to the language point of view. By recognizing these facts, it is a logical conclusion that the instanceOf semantics need to be defined here.

We have proven that these additional instantiation semantics provide a dual instanceOf for all model constructs. For OGML this is the case, and for each language added to OGML this is the case as well as for additional intensional and extensional models.

Evaluation of Research Objectives

The research objectives (Section 1.4) have largely been achieved:

- ✓ *(1a) - to choose an appropriate domain for RQ 1 and study its concepts*

A study of Ontology was performed within the scope of this project.

✓ *(1b) - to propose a modeling architecture that represents models in accordance with Ontology*

OGML includes OGMLX and together they provide a view on the whole modeling architecture. We propose this composition as our modeling architecture.

✓ *(1c) - to propose a metalanguage based on Ontology that includes means to capture instantiation semantics of modeling languages*

OGML was defined and proposed. It is based ontological constructs and can describe instanceOf semantics.

✓ *(2a) - to provide tool support for performing: language definition, model definition, import and export, check of model and language conformance*

This is included in the prototype except the import and export functionality. This prevents the current prototype to process and check large models drawn from real-world applications.

✓ *(2b) - creating a model query language to demonstrate the language independence of the modeling architecture and the tools*

An OCL Interpreter was realized and was shown to exhibit uniform treatment of all modeling constructs. The *LanguageAxis* expression makes this explicit.

✓ *(2c) - A case study of expressing UML while focusing on the instantiation of a complicated constructs like association*

The case study was conducted, the results were positive.

- *(2c) - A case study of expressing MOF to demonstrate support of multiple instantiation from model elements*

Due to a lack of time, we could not perform a case study on MOF.

Additional results

- We provided a proof of uniform model representation based on our (semi)formalized semantics definition of OGML
- Full OCL support for n-ary associations was realized as some researchers consider appropriate [78].

9.4 Discussion

Here we discuss our view on the results that was summarized in the previous sections. The most important results of OGML discussed here are: *uniform model handling*, *uniform model representation* and *Ontology-guided (meta)modeling*.

The Expense of Uniformity

We realized the two kinds of *uniformity* by increasing the expression power of the metalanguage. This property can be exploited in the following ways:

- Language independent model handling as demonstrated by a prototype of model navigation engine based on the OCL specification [73],
- Transformation definitions can become more language independent [62][65][18][60][53],
- Model reuse can be implemented in a language independent fashion, providing a good basis for model libraries [6][18],

- An OGML implementation can map the extensional structure directly on an existing data storage structure and directly adhere to basic requirements for any modeling architecture [9] "for free" as we demonstrated with a prototype implementation.

Due to a lack, we did not investigate whether OGML can express languages like OWL, RDF, MOF, UML (Packaging) and power types. Without this investigation, it is not known whether OGML traded *expressiveness for uniform model handling*.

Furthermore, the model handling requires extra lookups and operations on models for navigation. These operations even cascade to the level of the metalanguage, because the structural properties in the models are ontologically defined by OGML (see Subsection 4.3.3). Further investigation is needed to establish the complexity of navigation operations. Possible solution may be found in the work of Atzeni, Cappellari and Bernstein [12]. They propose a solution for schema independent data handling with the use of dictionaries in database systems. Their work was summarized in Section 8.4.

The Expense of Ontology

The domain of Ontology proved useable to support the modeling and metamodeling activities. With an example, we showed how metamodeling can be guided by ontological reasoning. We realize that this brings a potential overhead of educating (meta)modelers on ontological concepts or rather on the approach taken in Ontology. We however argue that the use of constructs in modeling can become less ambiguous if the concepts behind these constructs are well grounded in a systematic study. Furthermore, metamodeling is a specialized activity and requires some experience and knowledge anyhow.

The Use of Ontology

For using the four-category ontology, we can chose two approaches: a pragmatic one, where we choose to break the laws of the Ontology in order to express languages faithful to their specification or we could choose to be ontologically faithful. Currently we chose the more pragmatic approach and conceded to requirements coming from modeling. This approach has the potential to decrease the real-world relation of OGMLs constructs. Ontological imprecision would again reduce the consistent use of the constructs.

On the other hand, a complete refuge to ontological correctness can hardly yield a result that is usable in MDE. A balance between the two options needs to be found especially when OGML is extended as is discussed in the next section.

OGML Compared to Traditional Architectures

Traditional modeling architectures provide an underlying structure. OWL uses RDF, a data description language based on graphs, as its underlying structure. The mapping is provided by the RDF reification model [89][90]. In MOF, the underlying structure is the MOF-Object. By specialization, it is realized that all modeling elements are instances of it [71]. OGML is more like OWL; the manner in which model elements become instances of the underlying model is made explicit.

The underlying structure of OGML is OGMLX. It is an ontology-based model. Compared to RDF it preserves more information about the role of the model elements. RDF is represents all model elements as "Resource" in a graph structure, whereas models in OGMLX preserves the ontological category of a model element.

Replication of Concepts

Atkinson and Kühne state: "*the nested metalevels approach does not provide an answer to the "replication of concepts"* problem. OGML is a nested modeling architecture. In OGML, you could argue that some concepts are duplicated. The structure of OGMLX is based on the ontological square (see Figure 2-2). OGML also bases its constructs on the square. However, in our view these should be different constructs, because a distinction between them can be made on the base of their function in the model. The constructs of OGMLX give the absolute ontological nature of modeling elements. If, according to OGMLX, a model element is an individual, it is not instantiatable from any (language) perspective. OGML constructs, on the other hand, represent the (language) relative nature of model elements.

Furthermore, any further replication is prevented because OGML is limited to three modeling layers, which should prevent infinite replication of concepts. This excludes the conceptual replication that occurs because of the self-reflective definition of relations (Section 4.5). We have shown with the prototype that this problem can be overcome.

Another argument that could be raised is the limitation of three layers. This issue if discussed in the literature [11][34][33]. We think that three layers is the appropriate amount for two reasons: (1) from the absolute perspective, the metalanguage[19], there are only three layers (metametamodel, metamodel and model) and (2) we feel that representation power types can be done at the modeling layer[20].

Additional Requirements for the Modeling Architecture

In the beginning of MDA, UML was a self-described modeling language hardcoded in tools. When the demand came for support of multiple modeling languages, it was recognized that UML needed a relatively small subset of constructs to describe its own structure. Therefore, these constructs were isolated and put in MOF [72]. MOF provides the basis for language definitions paving the way for metamodeling [15].This resulted in a need for model based syntax definition. Several frameworks are currently at our disposal to provide a solution [51][52][38].

OGML introduces two explicit instanceOf relations in models. Both can be used for abstract syntax definition as noted in Section 9.2. For example, for an UML Object diagram, we could create a syntax based on the linguistic instanceOf. Enabling us to create and connect objects, slots and literals. We could also use the ontological instanceOf and create instances (of classes). Ideally, however, both are used. The linguistic instanceOf can be used for the visual

[19] By extending the metalanguage, we already treated it as the absolute perspective. We think the results prove that this perspective is indeed absolute.

[20] Whether this is true needs of course still to be proven as we show in future work.

syntax (Objects are represented as boxes, etc) and the ontological instanceOf can provide constraints on the AST. So that the created objects indeed, conform to some class in the model. An extra parameterization of these constraints would greatly enhance the metamodeling and modeling capabilities of the OGML architecture.

The uniform model handling results in relativity as shown in Section 5.4. The extra expression we introduced to the OCL Interpreter makes this explicit. We argue however that this relativity is wanted, and perhaps unavoidable. It provides the user with extra information (structural and ontological), most valuable for automation in MDE. Relating back to the definition we introduced for "model", we achieved to some extent to treat it as a truly multi-intensional artifact. However still more is possible, as we will see in the coming section about future work.

Meta-Muddle

The traditional modeling architectures are also under theoretical investigation. The problems found in it have some researcher led to refer to it as "meta-muddle" [83][37][32][82]. Although the research about the architecture has given some insights, we argue that some of it simply is not sound. An example comes from Atkinson and Kühne [11]. They argue that a modeling architecture needs to commit to either a *"language"* or a *"library"* metaphor for the underlying structure. In Section 4.8, we showed that OGMLX is both, depending on the perspective.

9.5 Future Work

In this section, we propose some possibilities for extending and researching OGML. Finally a recommendation is given. The status of OGML is best understood by seeing it as a prototype to demonstrate uniform model handling and the use of Ontology in metamodeling. Whether OGML can express languages like OWL, RDF, MOF, UML (Packaging) and constructs like power types needs investigation. Without this investigation, it is not known whether OGML traded *expressiveness for uniform model handling*.

Case Studies

The investigation into the expressiveness of OGML could start with a case study expressing RDF and MOF in OGML. We do not expect that OGML needs to be extended for this purpose. The results of the case study will show OGML's merits with regard to multi level transformation between the different languages.

By adding support for multiple instantiation to the prototype, a case study for OWL can be supported by OGML.

Improvements

Currently not all details of OGML have been defined. This is the list of OGML "to-do's", which could help improving OGML:

- Improve the representation and interpretation of literals. Currently literals are all treated the same (as strings). Find a solution for their data types. How to relate them to the different storage formats of integer, string, etc? The interpretation of literals needs to be

uniform across layers. Currently every layer defines its own Literal and the value must always be stored in Value.

- From the ontological perspective, individuals need to have identity. This identity is provided in ontology by the values of the properties of the individual. For pragmatic and implementation reasons this cannot be adhered to in modeling practices. An explicit identification mechanism needs to be in place:
  "In essence, use of identification attributes reflects the ontological premise that everything is unique (no two things possess the same set of properties)." [87]
- Answer the question of inheritance between different categories. Currently we advice against it because of ontological correctness; however for technical reasons it might be desirable. To support it more generically, we could make concrete *UniversalDefinitions* and *IndividualDefinitions.* Currently these are abstract constructs.
- Reflect the results of solving the aforementioned "to-do's" in the prototype implementation.

Language Completion

For the current status of OGML, a limited set of Ontological concepts where needed (notably the structural ones). If the design of OGML is to be extended for a broader set of modeling languages and constructs, than it will need to derive more and more constructs from Ontological concepts that carry more laws. This will however allow the ability to express an increasing set of modeling language features like: containment, packaging and profiles.

- Mereology is the study of the part-whole relation. Mereological constructs support containment in a modeling language.
- Define instantiation for languages and models. Currently each language is instantiated to a model construct. It is not defined in what manner the instantiated language constructs become the content of the model. Expressing this may require first a definition of containment and thus requires mereological constructs. The instanceOf relation is strongly associated with languages, because it is relative to the language. For this reason, language instantiation could be a complex thing to define.
- Laws could be introduced to constructs to support constraint definition on the models.
- A diagrammatic syntax for OGML could save efforts in metamodeling and especially thesis writing.

Recommendations

We feel the expressiveness of the current OGML has not sufficiently been established. A breath first approach will establish the usefulness of OGML to a greater extend while at the same time increasing the knowledge about it. Therefore, we consider the conduct of more case studies of first priority, before extending the language any further.

If the case studies succeed, an iterative development can take place cycling over extending the metalanguage and conducting new case studies.

Bibliography

[1] Álvarez, J. M., Evans, A., and Sammut, P. 2001. Mapping between Levels in the Metamodel Architecture. In *Proceedings of the 4th international Conference on the Unified Modeling Language, Modeling Languages, Concepts, and Tools* (October 01 - 05, 2001). LNCS, vol. 2185. Springer-Verlag, London, 34-46.

[2] Armstrong, D. M. 1989. *Universals: An opinionated introduction*. Boulder, CO: Westview.

[3] Atkinson, C. 1997. Meta-Modeling for Distributed Object Environments. In Proceedings of the 1st international Conference on Enterprise Distributed Object Computing (October 24 - 26, 1997). EDOC. IEEE Computer Society, Washington, DC, 90.

[4] Atkinson, C. 1997. Meta-Modeling for Distributed Object Environments. In *Proceedings of the 1st international Conference on Enterprise Distributed Object Computing* (October 24 - 26, 1997). EDOC. IEEE Computer Society, Washington, DC, 90.

[5] Atkinson, C., Kühne, T, Meta-level independent modeling. In *International Workshop Model Engineering (in Conjunction with ECOOP'2000)*. Springer Verlag, Cannes, France, June 2000.

[6] Atkinson, C., Kühne, T. 2002. Profiles in a strict metamodeling framework. *Sci. Comput. Program.* 44, 1 (Jul. 2002), 5-22.

[7] Atkinson, C., Kühne, T. 2002. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.* 12, 4 (Oct. 2002), 290-321

[8] Atkinson, C., Kühne, T. 2003. Calling a Spade a Spade in the MDA Infrastructure, In: *Proceedings of the Metamodeling for MDA First International Workshop*, York, UK, November 2003, 9-12.

[9] Atkinson, C., Kühne, T. Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5), pp. 36-41, 2003

[10] Atkinson, C., Kühne, T. The Essence of Multilevel Metamodeling. UML 2001: 19-33

[11] Atkinson, C., Kühne, T.: Concepts for Comparing Modeling Tool Architectures, In: Proceedings of theACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, MoDELS/UML 2005

[12] Atzeni, P., Cappellari, P., Bernstein, P., A. A multilevel dictionary for model management, in: ER 2005, LNCS, vol. 3716, 2005, pp. 160-175

[13] Atzeni, P., Cappellari, P., Torlone, R., Bernstein, P. A., and Gianforme, G. 2008. Model-independent schema translation. *The VLDB Journal* 17, 6 (Nov. 2008), 1347-1370

[14] Beck, K. 1999. Embracing Change with Extreme Programming. *Computer* 32, 10 (Oct. 1999), 70-77.

[15] Bézivin, J. 2001. From Object Composition to Model Transformation with the MDA. In *Proceedings of the 39th international Conference and Exhibition on Technology of Object-Oriented Languages and Systems (Tools39)* (July 29 - August 03, 2001). TOOLS. IEEE Computer Society, Washington, DC, 350.

[16] Bézivin, J. In Search of a Basic Principle for Model Driven Engineering. UPGRADE V(2), Novótica, April 2004

[17] Bézivin, J. On the unification power of models. Software and System Modeling 4(2): 171-188 (2005)

[18] Bézivin, J., Gerbé, O. 2001. Towards a Precise Definition of the OMG/MDA Framework. In Proceedings of the 16th IEEE international Conference on Automated Software Engineering (November 26 - 29, 2001). Automated Software Engineering. IEEE Computer Society, Washington, DC, 273.

[19] Bézivin, J., Jouault, F., Valduriez, P., On the Need for Megamodels, OOPSLA & GPCE, Workshop on best MDSD practices, Vancouver, Canada, 2004

[20] Bezivin, J., Kurtev, I: Model-based Technology Integration with the Technical Space Concept. In: Metainformatics Symposium. Springer-Verlag.

[21] Bézivin, J., Lemesle, R. 1998. Ontology-Based Layered Semantics for Precise OA&D Modeling. In *Proceedings of the Workshops on Object-Oriented Technology* (June 09 - 13, 1997). LNCS, vol. 1357. Springer-Verlag, London, 151-154.

[22] Bloomfield, L. 1933. Language. Holt, Rinehart and Winston, New York.

[23] Bostock, S., July 2007. The four-category ontology: A metaphysical foundation for natural science - by e.j. lowe. Philosophical Books 48 (3), 274-277.

[24] Bowers, S., Delcambre, L. On modeling conformance for flexible transformation over data models. In Proceedings of the Workshop on Knowledge Transformation for the Semantic Web at the 15th European Conference on Artificial Intelligence (KTSW-2002), Lyon, France, 2002

[25] Chomsky, N., Three models for the description of language. IEEE Transactions on Information Theory. v2 i3. 113-124

[26] Degen, W., Heller, B., Herre, H., Smith, B. 2001. GOL: toward an axiomatized upper-level ontology. In Proceedings of the international Conference on Formal ontology in information Systems - Volume 2001 (Ogunquit, Maine, USA, October 17 - 19, 2001). FOIS '01. ACM, New York, NY, 34-46.

[27] Duddy, K. 2002. UML2 must enable a family of languages. *Commun. ACM* 45, 11 (Nov. 2002), 73-75.

[28] Eclipse Modeling Framework (EMF). Available at http://www.eclipse.org/emf/

[29] Evans, A. and Kent, S. 1999. Core meta-modelling semantics of UML: The pUML approach. In Proceedings of UML'99. Lecture Notes in Computer Science, vol. 1723. Springer-Verlag, New York, 140--155.

[30] Evermann, J., Wand, Y. "Towards Ontologically Based Semantics for UML Constructs, In Proceedings of Conceptual Modeling - ER 2001 - 20th International Conference on Conceptual Modeling, Yokohama, Japan, November 27-30, 2001, Berlin, Heidelberg 2001, pp. 354-367

[31] Falkenberg, E., Verrijn-Stuart, A., Voss, K., Hesse, W., Lindgreen, P., Nilsson, B., Oei, J., Rolland, C., and Stamper, R. a. A Framework of Information Systems Concepts. The FRISCO report. 1998

[32] Favre, J.M. Megamodeling and Etymology, Dagstuhl Seminar Proceedings 05161 on Transformation Techniques in Software Engineering, DROPS, http://drops.dagsthul.de

[33] Geisler, R., Klar, M., and Pons, C. 1998. Dimensions and dichotomy in metamodeling. In *Proceedings of the Third BCS-FACS Northern Formal Methods Workshop* (September). Springer-Verlag, New York.

[34] Gitzel, R., Hildenbrandt, T.: A Taxonomy of Metamodel Hierachies - Working Paper 1-05. Available at http://www.wifo.uni-mannheim.de/~gitzel/publications/taxonomy.pdf. (2005).

[35] Gödel, K.: On formally undecidable statements of Principia Mathematica, and related systems. Monatshefte für Mathematik und Physik 38, 173-198 (1931) (translated by A. Meltzer, introduced by A. Braithwaite, Dover Publications).

[36] Gogolla, M. and Richters, M. 2002. Expressing UML Class Diagrams Properties with OCL. In Object Modeling with the Ocl, the Rationale Behind the Object Constraint Language T. Clark and J. Warmer, Eds. Lecture Notes In Computer Science, vol. 2263. Springer-Verlag, London, 85-114.

[37] Gogolla, M., J.-M. Favre and F. Büttner, On Squeezing M0, M1, M2, and M3 into a Single Object Diagram, Technical Report LGL-REPORT-2005-001, Ecole Polytechnique Fédérale de Lausanne (2005)

[38] Greenfield, J., Short, K., Cook, S., Kent, S., Software Factories, Wiley, ISBN 0-471-20284-3, 2004.

[39] Guarino, N. and Welty, C. A. 2000. A Formal Ontology of Properties. In *Proceedings of the 12th European Workshop on Knowledge Acquisition, Modeling and Management* (October 02 - 06, 2000). R. Dieng and O. Corby, Eds. Lecture Notes In Computer Science, vol. 1937. Springer-Verlag, London, 97-112.

[40] Guarino, N., "Formal Ontology and Information Systems," in *Formal Ontology in Information Systems*, N. Guarino, Ed. Amsterdam, Netherlands: IOS Press, 1998

[41] Guizzardi, G. (2005). *Ontological Foundations for Structural Conceptual Models*, Telematica Instituut Fundamental Research Series, Vol. 015. Enschede: Telematica Instituut

[42] Guizzardi, G., Ferreira Pires, L., van Sinderen, M. An Ontology-Based Approach for Evaluating the *Domain Appropriateness* and *Comprehensibility Appropriateness* of Modeling Languages. MoDELS 2005: 691-705

[43] Guizzardi, G., Herre, H., and Wagner, G. 2002. On the General Ontological Foundations of Conceptual Modeling. ER 2002: 65-78

[44] Guizzardi, G., Herre, H., Wagner, G., 2002. Towards ontological foundations for uml conceptual models. In: On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002. Springer-Verlag, London, UK, pp. 1100-1117.

[45] Guizzardi, G.: Ontological Foundations for Structural Conceptual Models. Telematica Instituut, Enschede (2005)

[46] Harel, D., Rumpe, B., August 2000. Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. Tech. rep., Mathematics & Computer Science, Weizmann Institute Of Science, Mathematics & Computer Science, Weizmann Rehovot, Israel.

[47] Henderson-Sellers, B., Gonzalez-Perez, C.: 2005. Connecting Powertypes and Stereotypes. J. Object Technol. 4(7), 83–96

[48] Hull, R., King, R. 1987. Semantic database modeling: survey, applications, and research issues. *ACM Comput. Surv.* 19, 3 (Sep. 1987), 201-260.

[49] Intentional Software. Available at http://www.intentsoft.com

[50] Jouault, F., Bézivin, J. KM3: a DSL for Metamodel Specification. FMOODS 2006, Bologna, Italy, 14-16 June 2006

[51] Jouault, F., Bézivin, J., and Kurtev, I. 2006. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th international Conference on Generative Programming and Component Engineering* (Portland, Oregon, USA, October 22 - 26, 2006). GPCE '06. ACM, New York, NY, 249-254.

[52] Jouault, F., Bézivin, J., Consel, C., Kurtev, I., and Latry, F. Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages. In *Proceedings of the 1st ECOOP Workshop on Domain-Specific Program Development (DSPD'06)*, Nantes, France, July 2006.

[53] Kappel, G, Kapsammer, E, Kargl, H, Kramler, G, Reiter, T, Retschitzegger, W, Schwinger, W, and Wimmer, M : Lifting metamodels to ontologies - a step to the semantic integration of modeling languages. In: ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MODELS'06). 2006

[54] Kelly, S., Tolvanen, J.-P. Domains-specific modeling: enabling full code generation. Wiley-IEEE Computer Society, 2008

[55] Kent, S. Model Driven Engineering. In Proceedings of IFM2002, LNCS 2335, Springer, 2002

[56] Kleppe, A., G., Warmer, J., and Bast, W. 2003 *MDA Explained: the Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc.

[57] Kolovos, D., S., Paige, R., F., Polack, F.: Aligning OCL with Domain-Specific Languages to Support Instance-Level Model Queries. ECEASST 5, 2006

[58] Kruchten, P. 2003 *The Rational Unified Process: an Introduction*. 3. Addison-Wesley Longman Publishing Co., Inc.

[59] Kühne, T., 2002. Matters of (meta-) modeling-the role of metamodeling. In: in MDA", International Workshop in Software Model Engineering (in conjunction with UML'02.

[60] Kurtev, I. 2008. Application of Reflection in Model Transformation Languages. In *Proceedings of the 1st international Conference on theory and Practice of Model Transformations* (Zurich, Switzerland, July 01 - 02, 2008). A. Vallecillo, J. Gray, and A. Pierantonio, Eds. Lecture Notes In Computer Science, vol. 5063. Springer-Verlag, Berlin, Heidelberg, 199-213.

[61] Kurtev, I. 2008. State of the Art of QVT: A Model Transformation Language Standard. In *Applications of Graph Transformations with industrial Relevance: Third international Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and invited Papers*, LNCS, vol. 5088. Springer-Verlag, Berlin, Heidelberg, 377-393.

[62] Kurtev, I. Adaptability of Model Transformations. PhD thesis, University of Twente, 2005. ISBN 90-365-2184-X

[63] Kurtev, I. Metamodels: Definitions of Structures or Ontological Commitments? Workshop on TOWERS of models. Collocated with TOOLS Europe. 2007

[64] Kurtev, I., Bézivin, J., Aksit, M. 2002. Technological Spaces: An Initial Appraisal. In *Proceedings of the EWI-SE: Software Engineering* (Irvine, USA, 1 Nov 2002) 1-6

[65] Kurtev, I., van den Berg, K. MISTRAL: A Language for Model Transformations in the MOF Meta-modeling Architecture. MDAFA 2004: 139-158

[66] Loux, M.J. The problem of universals. In Metaphysics: contemporary readings. M.J. Loux (ed.). Routledge, 2001

[67] Milton, S., and Kazmierczak, E. (2004), "An Ontology of Data Modeling Languages: A Study Using a Common-Sense Realistic Ontology," Journal of Database Management, Vol. 15, No.2, pp. 19-38.

[68] Mylopoulos, J., Information Modeling in the Time of the Revolution. Information Systems, Vol. 23, 1998

[69] Object Management Group (OMG), Common Warehouse Metamodel (CWM) Specification, Version 1.1, formal/2003-03-02, 2003

[70] Object Management Group (OMG), MDA Guide Version 1.0.1, Doc. omg/2003/06/01, 2003

[71] Object Management Group (OMG), Meta object facility (MOF) 2.0 Query/View/Transformation Specification, Final Adopted Specification, ptc-07/07/07, 2007

[72] Object Management Group (OMG), Meta object facility (MOF) core specification, Version 2.0 formal/06/01/01, 2006

[73] Object Management Group (OMG), OCL 2.0 Specification, Version 2.0, ptc/2005/06/06, 2005

[74] Object Management Group (OMG), UML 2.0 Infrastructure Specification. www.omg.org, Sept. 2003.

[75] Object Management Group (OMG), UML 2.0 Specification, Version 2.0, ptc/2008/05/07, 2008

[76] Odell, J. Power Types. 1994. JOOP 7(2): 8-12

[77] Quine, W. V. O. 1969. Ontological relativity. In W. V. O. Quine (Ed.), Ontological relativity and other essays (pp. 26-68). New York: Columbia Univ. Press.

[78] Richters, M., Gogolla, M., On the need for a precise OCL semantics. In Proceedings of OOPSLA Workshop "Rigorous Modeling and Analysis with the UML: Challenges and Limitations". Colorado State University, Fort Collins, Colorado, 1999.

[79] Russell, B., 1946. The History of Western Philosophy, Allen and Unwin, London. Unwin, 1985

[80] Russell, Bertrand: "Letter to Frege". In : in van Heijenoort, Jean, From Frege to Gödel 124-125. Cambridge, Mass.: Harvard University Press (1967) 124-125

[81] Snyder, A. 1986. Encapsulation and inheritance in object-oriented programming languages. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (Portland, Oregon, United States, September 29 - October 02, 1986). N. Meyrowitz, Ed. OOPLSA '86. ACM, New York, NY, 38-45.

[82] Soden, M.: Operational Semantics for MOF Metamodels. Working version. Available at http://www.metamodels.de/publications.html

[83] Solberg, A., France, R., Reddy, R., "Navigating the MetaMuddle," *Proceedings of the 4th Workshop in Software Model Engineering*, Montego Bay, Jamaica, 2005, Available at http://www.planetmde.org/wisme-2005/ NavigatingTheMetaMuddle.pdf

[84] Sowa J., F. Ontology, Metadata, and Semiotics, Proceedings of the Linguistic on Conceptual Structures: Logical Linguistic, and Computational Issues, p.55-81, August 14-18, 2000

[85] Sowa, J. F. 1992. Conceptual graphs summary. In *Conceptual Structures: Current Research and Practice*, T. E. Nagle, J. A. Nagle, L. L. Gerholz, and P. W. Eklund, Eds. Ellis Horwood Series In Workshops. Ellis Horwood, Upper Saddle River, NJ, 3-51.

[86] Sowa, J. F. 2000. Knowledge Representation: Logical, Philosophical, and Computational Foundations. Brooks/Cole Publishing.

[87] Wand, Y., Storey, V.C., Weber, R. An Ontological Analysis of the Relationship Construct in Conceptual Modeling. ACM Trans. Database Syst. 24(4): 494-528 (1999)

[88] Watt, A.,D., Brown, D., F., "Programming Language Processors in Java. Compilers and Interpreters", Preatice Hall, 2000.

[89] World Wide Web Consortium, 'Resource Description Framework (RDF)', Available at http://www.w3.org/RDF/. W3C Recommendation 10 February 2004

[90] World Wide Web Consortium, 'Web Ontology Language (OWL)', Available at http://www.w3.org/2004/OWL/. W3C Recommendation 10 February 2004

Appendix A – Concrete Syntax of OGML

This appendix describes the grammar of the meta-language presented in Chapter 4. To represent the grammar we use Extended Backus-Naur Form (EBNF). Non-terminals are in bold text and have a Capital first letter. Terminals are quoted. Identifiers, consist of any sequence of characters without a space. They are represented starting with a small letter.

A.1 EBNF

The syntax of EBNF is expressed in itself as follows (letter and all-characters are not fully expanded):

<production-rule> ::= <non-terminal> < ::= > <sequence>
< sequence> ::= <element> | <element> < sequence>
<element> ::= < quantifier > | <choice>
<choice> ::= < sequence > <|> < sequence>
<quantifier> ::= <expression> * | <expression> + | <expression> ?
<expression> ::= <atom> | (< sequence>)
<atom > ::= <non-terminal> | <identifier> | <terminal>
<non-terminal> ::= <capital> | <all-characters> <non-terminal>
< identifier > ::= ” <all-characters> ”
< terminal > ::= <letter> | <letter> <all-characters>

The grammar does not fully expand the included OCL definition. The syntax of OCL can be found in [73]. For more information on conformance to the OCL standard, the reader should consult Appendix C *“OCL* Interpreter and Metamodel”.

In the following subsections, the syntax is presented. The same partitioning of the language is used as in Chapter 4 to enable easy referencing.

A.2 Language Constructs

LanguageDefinition ::= "Language" name "{"
LanguageContent *
"}"

LanguageContent ::= **Definition | Relations | GeneralizationRelation**

Definition ::= **UniversalDefinition | UniversalDefinition**

UniversalDefinition ::= **SubstantialDefinition | MomentDefinition**

IndividualDefinition ::= **ObjectDefinition | PropertyDefinition**

SubstantialDefinition ::= "SubstantialDefinition" name **("extends" (***definition* **(","***definition***)*))?**
"{"
Attribute *
"}"

MomentDefinition ::= "MomentDefinition" name **("extends" (***definition* **(","***definition***)*))?**
"{"
Attribute *
CharacterizationRelation +
"}"

DataTypeDefinition ::= "DataTypeDefinition" name **("extends" (***definition* **(","***definition***)*))?**
"{"
Attribute *

```
                                    "}"

ObjectDefinition            ::=     "ObjectDefinition" name ("extends" (definition ("," definition)*) )?
                                    "{"
                                            Attribute *
                                    "}"

PropertyDefinition          ::=     "PropertyDefinition" name ("extends" (definition ("," definition)*) )?
                                    "{"
                                            Attribute *
                                            InherenceRelation +
                                    "}"
```

A.3 Relational Constructs

```
Attribute                   ::=     "attribute" name "[" lower "-" upper "]" ":" definition ("," definition)* ";"

CharacterizationRelation    ::=     "characterization"
                                            universalDefinitionRole ":" ownerDefinition ( "," ownerDefinition )*
                                            "momentDefinitionRole"   "[" lower "-" upper "]" momentDefinitionRole ";"

InherenceRelation           ::=     "dependsOn" ( propertyBearer ("," propertyBearer)*)
                                    "role" "=" role "[" lower "-" upper "]"  ";"
```

A.4 Ontological Perspective Constructs

```
Relations                   ::=     "Relations" name "{"
                                             instanceOfRelations
                                    "}"

InstanceOfDefinition        ::=     ("abstract")?  (definingConceptIdentifier ":") ?  definition  "->"
                                    ( sequenceIdentifier ":" "[" ) ?
                                            (instanceIdentifier ":") ? conformingDefinition
                                     ( "]" ) ?
                                     "{"
                                            CharacterizationInstantiation *
                                            AttributeFunction *
                                     "}" ( "when" "(" ExpressionInOcl ")" ) ?

CharacterizationInstantiation ::=   characterizationRole "->" momentRole ";"

AttributeFunction           ::=     characterizationRole
                                    "{"
                                            "naming" name "<-" ExpressionInOcl ";"
                                            "valuing" "[" lower ".." upper "]" ExpressionInOcl ";"
                                            "typing" ExpressionInOcl  ";"
                                    "}" ( "where" "(" ExpressionInOcl ")" ) ?
```

A.5 Generalization and Specialization Constructs

```
GeneralizationRelation      ::=     "GeneralizationRelation" name
                                    "{"
                                            "generalConcept" "=" generalConcept  ("," generalConcept )*  ";"
                                            "specializedConcept" "=" specializedConcept ("," specializedConcept )* ";"
                                            "parentMultiplicity" "="  "[" lower "-" upper "]"  ";"
                                            "childMultiplicity"  "="  "[" lower "-" upper "]"  ";"
                                            "generalConceptRole" "=" generalConceptRole  ";"
                                            "specializedConceptRole" "=" specializedConceptRole  ";"
                                    "}"
```

A.6 Other Constructs

```
OGMLDataType                ::=     "OGMLDataType" name  ("extends" (definition ("," definition)*) ) ?
                                    "{"
                                            Attribute *
                                    "}"
```

Class ::= "Class" name ("extends" (*definition* ("," *definition*)*)) ?
"{"
Attribute *
"}"

ExpressionInOcl ::= bodyExpression;

A.7 Symbol Table Creation

The parsed syntax of OGML results in a tree. In order to create the graph structure of the abstract syntax that was presented in UML diagrams in Chapter 4, some identifiers are matched by name in a symbol table [88]. Here the identifiers are matched by-name other parts of the parsed tree. Here we express these relations in terms of the concrete syntax that was just defined. The following syntax is used:

< reference-rule> ::= <reference> → <non-terminal> . <identifier-in-non-terminal-production-rule>
<reference> ::= <identifier>

definition → **Definition.**name

ownerDefinition → **UniversalDefinition.**name

propertyBearer → **PropertyDefinition.**name

definition → **UniversalDefinition.**name

conformingDefinition → **Definition.**name

generalConcept → **Definition.**name

specializedConcept → **Definition.**name

Appendix B – OGML Definition

In this appendix, we present OGML as it is expressed in its own syntax. The syntax is found the previous appendix. Like the grammar, this definition does not fully expand the included OCL definition. Whenever an identifier collides with it a keyword of the language it has to be escaped, this is done by parenthesis. Escaped identifiers are also shown in quotes. The syntax of OCL can be found in [73].

OGML Definition

```
1:  Language OGML {
        SubstantialDefinition Definition extends Classifier {
                attribute name : "String";
        }

5:      SubstantialDefinition UniversalDefinition extends Definition {}
        SubstantialDefinition "SubstantialDefinition" extends UniversalDefinition {}
        SubstantialDefinition "MomentDefinition" extends UniversalDefinition {}
        SubstantialDefinition "DataTypeDefinition" extends UniversalDefinition {}

10:     SubstantialDefinition IndividualDefinition extends Definition {}
        SubstantialDefinition "ObjectDefinition" extends IndividualDefinition {}
        SubstantialDefinition "PropertyDefinition" extends IndividualDefinition {}

        SubstantialDefinition "Class" extends Definition {}
15:     DataTypeDefinition "OGMLDataType" extends Definition {}

        SubstantialDefinition LanguageDefinition {
                attribute definitions [*] : Definition, "Relations", "GeneralizationRelation";
        }
20:
        MomentDefinition Attribute {
                attribute name : "String";
                attribute range  [1-*] : Definition;
                attribute lower : "Integer";
25:             attribute upper : "Integer";
                characterization " owner " : Definition momentDefinitionRole [*] " attributes ";
        }

        MomentDefinition CharacterizationRelation {
30:             attribute id : "String";
                attribute lower : "Integer";
                attribute upper : "Integer";
                attribute "momentDefinitionRole" : "String";
                attribute "universalDefinitionRole" : "String";
35:             characterization dependentDefinition : MomentDefinition momentDefinitionRole [1-*]  dependency;
                characterization ownerDefinition : UniversalDefinition momentDefinitionRole [*]  feature;
        }

        MomentDefinition InherenceRelation {
40:             attribute lower : "Integer";
                attribute upper : "Integer";
                attribute "role" : "String";
                characterization " property " : PropertyDefinition momentDefinitionRole [1-*] " inherenceRelation ";
                characterization " propertyBearer " : Definition momentDefinitionRole [*] " properties ";
45:     }

        MomentDefinition InstanceOfDefinition {
                attribute isAbstract : "Boolean";
                attribute instanceIdentifier[0-1] : "String";
50:             attribute sequenceIdentifier[0-1] : "String";
                attribute definingConceptIdentifier[0-1] : "String";
                attribute condition[0-1] : Expression;
                attribute characterizationInstantiations[*] : CharacterizationInstantiation;
                attribute attributeFunctions[*] : AttributeFunction;
55:             characterization definition : UniversalDefinition momentDefinitionRole [*] instanceOfRelation;
                characterization conformingDefinition : Definition momentDefinitionRole [*] instanceOf;
        }
```

```
MomentDefinition "GeneralizationRelation" {
        attribute name : "String";
        attribute "generalConceptRole" : "String";
        attribute "specializedConceptRole" : "String";
        attribute "generalConceptLower" : "Integer";
        attribute "generalConceptUpper" : "Integer";
        attribute "specializedConceptLower" : "Integer";
        attribute "specializedConceptUpper" : "Integer";
        characterization "generalConcept" : Definition momentDefinitionRole [*] "specializations";
        characterization "specializedConcept" : Definition momentDefinitionRole [*] "generalizations";
}

GeneralizationRelation OGMLGeneralization {
        generalConcept = Definition, "Class", "OGMLDataType";
        specializedConcept = Definition, "Class", "OGMLDataType";
        parentMultiplicity = *;
        childMultiplicity = *;
        generalConceptRole = "extends";
        specializedConceptRole = "extendedBy";
}

Class "OclAny" {}
OGMLDataType "String" extends "OclAny" {}
OGMLDataType "Integer" extends "Double" {}
OGMLDataType "Boolean" extends "OclAny" {}
OGMLDataType "Double" extends "OclAny" {}

Relations OGMLInstanceOfDefinition {

  ld : LanguageDefinition -> mm : MetaModel {
        definitions -> contents;
  }

  abstract Definition -> PropertiesElement {
        attributes -> properties;
        properties -> properties;
        generalizations -> properties;
        specializations -> properties;
        instanceOf -> instanceOf;
  }

  sd : UniversalDefinition -> su : InstantiatableElement {
        feature -> properties;
        instanceOfRelation -> instantiatedTo;
  }

  md : "MomentDefinition" -> mu : MomentUniversal {
        dependency -> properties;
  }

 "PropertyDefinition" -> XObject {}
 "SubstantialDefinition" -> SubstantialUniversal {}
 "DataTypeDefinition" -> SubstantialUniversal {}
 "ObjectDefinition" -> XObject {}
 "Class" -> XObject {}
 "OGMLDataType" -> Literal {}
 InstanceOfDefinition -> InstanceOfProperty {}

  a : Attribute -> p : Property {
        attributes {
                naming name <- a.name;
                valuing [a.lower .. a.upper] p.value ;
                typing a.range;
        }
  }

  i : InherenceRelation -> p : Property {
        properties {
                naming name <- i."role";
                valuing [i.lower .. i.upper] p.value;
```

```
                typing i.property;
            }
        }

        c : CharacterizationRelation -> p1 : Property {
            feature {
                naming name <- c."momentDefinitionRole";
                valuing [c.lower .. c.upper] p1.value;
                typing c.dependentDefinition;
            }
        }
        c : CharacterizationRelation -> p2 : Property {
            dependency {
                naming name <- c."universalDefinitionRole";
                valuing [0 .. -1] p2.value;
                typing c.ownerDefinition;
            }
        }

        g : "GeneralizationRelation" -> p1 : Property {
            generalizations {
                naming name <- g."generalConceptRole";
                valuing [g.generalConceptLower .. g.generalConceptUpper] p1.value;
                typing g."generalConcept";
            }
        }
        g : "GeneralizationRelation" -> p2 : Property {
            specializations {
                naming name <-g."specializedConceptRole";
                valuing [g.specializedConceptLower .. g.specializedConceptUpper] p2.value;
                typing g."specializedConcept";
            }
        }
    }

    Class "Relations" {
        attribute name : "String";
        attribute instanceOfRelations[*] : InstanceOfDefinition;
    }

    Class CharacterizationInstantiation {
        attribute characterizationRole : "String";
        attribute momentRole : "String";
    }

    Class AttributeFunction {
        attribute characterizationRole : "String";
        attribute momentRole [0-1] : "String";
        attribute name : "String";
        attribute lower : Expression;
        attribute upper : Expression;
        attribute identifier : Expression;
        attribute value : Expression;
        attribute type : Expression;
        attribute condition [0-1] : Expression;
    }


    SubstantialDefinition ModelElement {}
    SubstantialDefinition PropertiesElement extends ModelElement {}
    SubstantialDefinition IdentifiableElement extends PropertiesElement {}
    SubstantialDefinition InstantiatableElement extends IdentifiableElement {
        attribute instantiatedTo[*] : InstanceOfProperty;
    }
    SubstantialDefinition ModelContent extends IdentifiableElement {
        attribute container : ModelContent;
    }

    SubstantialDefinition "Model" extends IdentifiableElement {
     attribute contents [*] : ModelContent;
    }
```

```
        SubstantialDefinition "MetaModel" extends "Model", InstantiatableElement {}

        SubstantialDefinition "SubstantialUniversal" extends InstantiatableElement, ModelContent {}
205:    SubstantialDefinition "MomentUniversal" extends InstantiatableElement, ModelContent  {}

        ObjectDefinition XObject extends IdentifiableElement, ModelContent {}
        ObjectDefinition Literal extends PropertiesElement {
                attribute "value" [*] : "String";
210:    }

        PropertyDefinition Property extends ModelElement {
                attribute name : "String";
                attribute "value"[*] : PropertiesElement;
215:            dependsOn IdentifiableElement role = "properties" multiplicity = * ;
        }

        PropertyDefinition InstanceOfProperty extends ModelElement {
                attribute "value"[*] : "Model", "MomentUniversal", SubstantialUniversal;
220:            attribute "language" : "String";
                dependsOn PropertiesElement, Property, InstanceOfProperty role = "instanceOf" multiplicity = *;
        }


225:    Class ExpressionInOcl extends OpaqueExpression {
                attribute bodyExpression : OclExpression;
                attribute resultVariable[0-1] : VariableDeclaration;
                attribute contextVariable[0-1] : VariableDeclaration;
                attribute parameterVariable[*] : VariableDeclaration;
230:    }

        ... OCL ...

    }
```

Appendix C – OCL Interpreter and Metamodel

This appendix includes the design of the OCL implementation. The OCL specification can be found in [73]. The OCL interpreter implements three base functionalities: OCL expression parsing, type checking and evaluation. For these functionalities, three class hierarchies are used: OclExpression, Type and Value. Each of them conforms to the abstract syntax trees from the OCL specification [73]. Some additions were made to make OCL language aware, as described in Chapter 7. Therefore the OCL metamodel is also provided here.

C.1 High- Level Design

The three class hierarchies in OCL are interrelated in an interesting way. Types are first class values in this implementation thus become a direct instance of the Value hierarchy. Furthermore, types play a dual role of type and expression, because they are one-to-one mapped to the syntax. Operations allow transformations between the hierarchies; they are displayed as dashed arrows in the diagram.

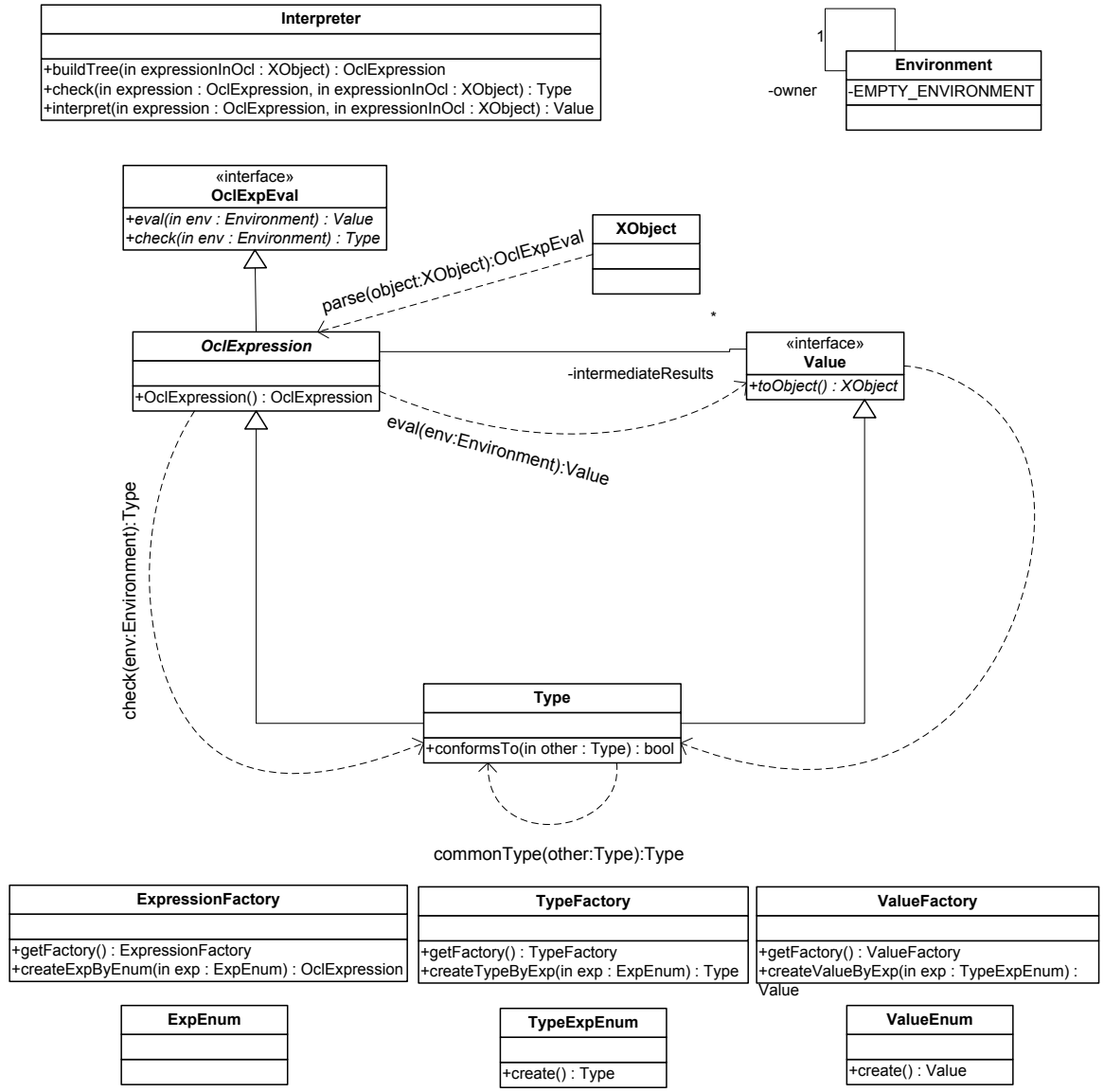


Figure C-9-1 - A high-level design of the OCL interpreter

C.2 Metamodel of Expression Hierarchy

Figure C-9-3 shows the OCL expression hierarchy.

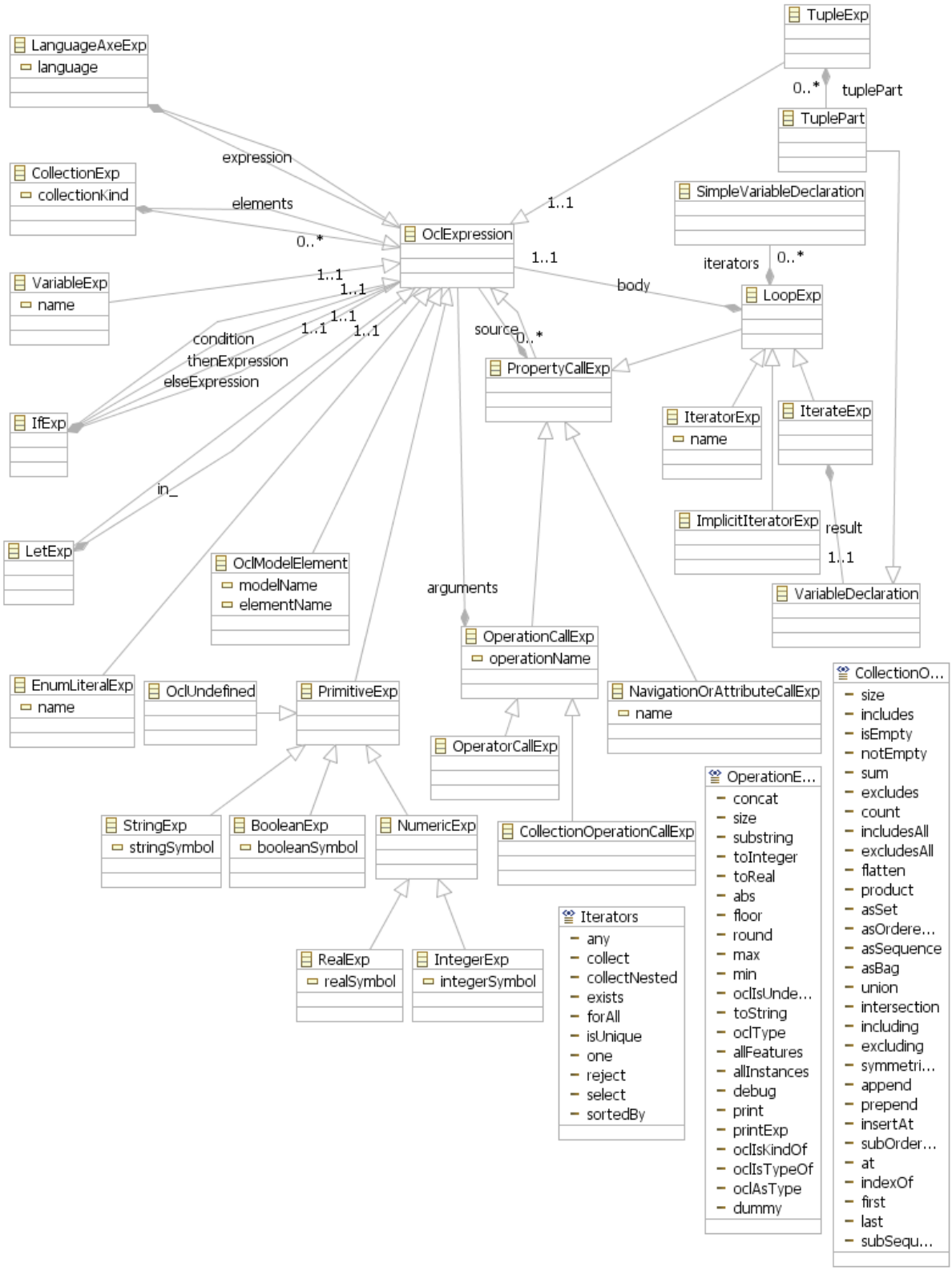


Figure C-9-2 - OCL Interpreter's Expression Hierarchy

C.3 Metamodel of Type Hierarchy

Figure C-9-3 shows the adapted OCL type hierarchy. An explicit *ModelElementType* was introduced as type and first-order value to support the language dependent notion of model element types.

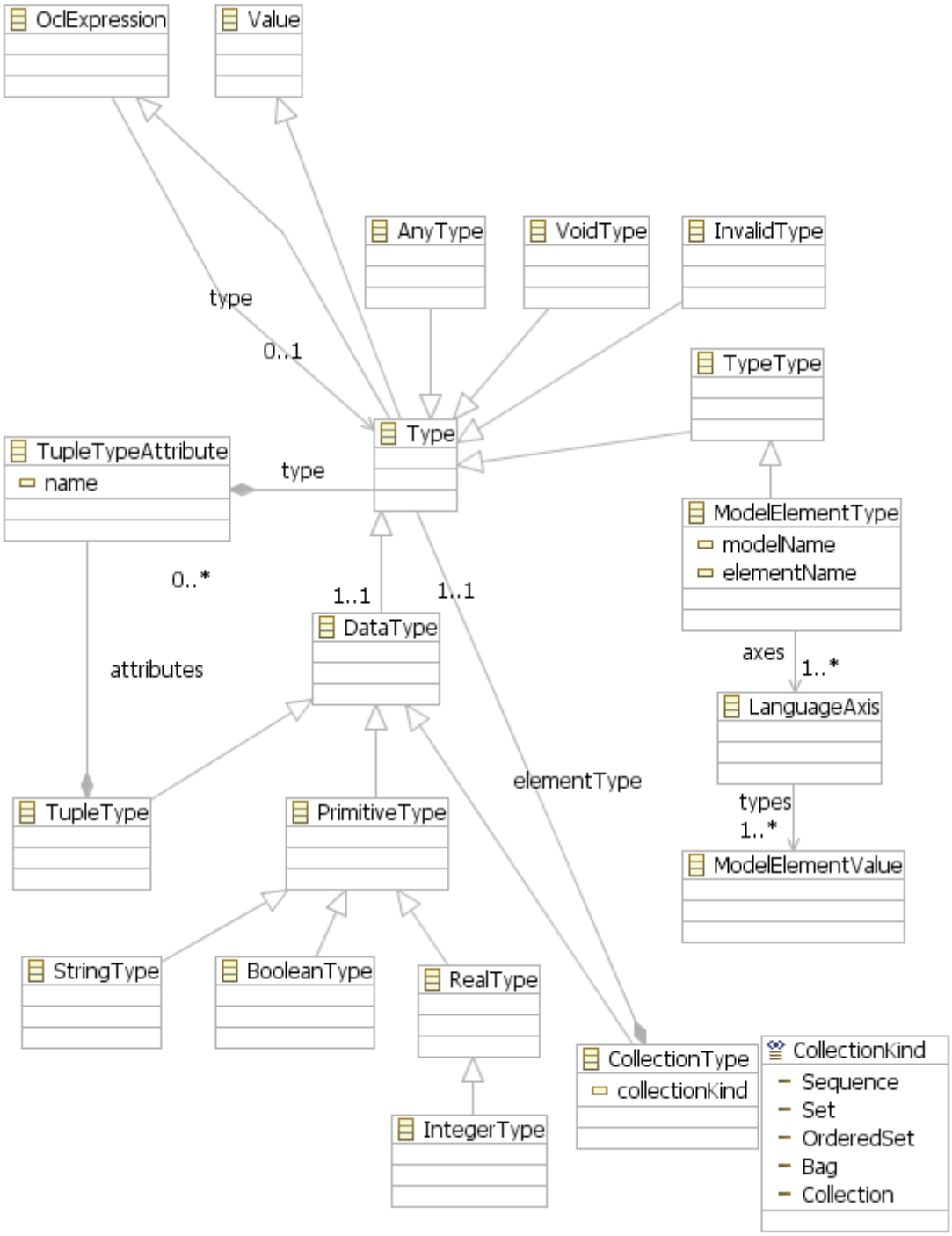


Figure C-9-3 - OCL Interpreter's Type Hierarchy

Index