Universidade Federal do Espírito Santo

Roberto Carraretto

A Modeling Infrastructure for OntoUML

Vitória - ES, Brazil

July, 2010

Roberto Carraretto

A Modeling Infrastructure for OntoUML

Monografia apresentada ao curso de Ciência da Computação do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. João Paulo Andrade Almeida

Vitória - ES, Brazil

July, 2010

RESUMO

Essa monografia provê uma infra-estrutura para uma linguagem de modelagem filosoficamente e cognitivamente bem-fundada, chamada OntoUML, que é uma extensão da *Unified Modeling Language* (UML). Primeiramente, revisamos a sintaxe abstrata da OntoUML (o metamodelo da linguagem) para torná-la adequada às regras de extensão da UML 2. Em seguida, desenvolvemos um editor gráfico para que os usuários possam manipular e criar modelos OntoUML. Na sequência, criamos um metamodelo de referência que é dedicado exclusivamente à sintaxe abstrata e contém restrições para garantir que um modelo esteja sintaticamente correto. O objetivo é criar um metamodelo que não depende de nenhuma tecnologia de edição gráfica e que seja o centro de atividades tais como transformações de modelo que relacionam OntoUML com outras linguagens. Para conectar a edição gráfica com a validação sintática de modelos, fornecemos uma transformação do metamodelo do editor gráfico para o metamodelo de referência. Adicionalmente, provemos uma transformação similar relacionando um editor gráfico definido em um trabalho anterior com o nosso metamodelo de referência. Portanto, nesse trabalho provemos uma infra-estrutura para OntoUML que auxilia a edição, validação sintática e transformação de modelos.

ABSTRACT

This work provides an infrastructure for a philosophically and cognitively well-founded modeling language, named OntoUML, which is an extension of the Unified Modeling Language (UML). First, we review the abstract syntax of OntoUML (the language's metamodel) in order to accommodate the rules for extending UML 2. Then, we design a graphical editor in order to allow users to manipulate the concrete syntax and create OntoUML models. In the sequel, we design a reference metamodel that is solely dedicated to abstract syntax elements and contains constraints for ensuring that a model is syntactically correct. The goal is to make a metamodel that does not depend on any specific graphical editor technology and is the core of activities such as model transformations that relate OntoUML to other languages. In order to connect the graphical editing with syntax validation, we provide a transformation from our graphical editor's metamodel to the reference metamodel. Moreover, we provide a similar transformation relating a graphical editor defined in a previous work with the reference metamodel. Thus, we provide a modeling infrastructure for OntoUML that assists the editing, syntax validation and transformation of models.

CONTENTS

1 INTRODUCTION

Information is an abundant, ubiquitous and precious resource. Most of the information available to us is in computer-based forms, such as files and databases. Thus, the task of computer scientists is to develop theories, tools and techniques for managing this information and making it useful. To use information, one needs to represent it, capturing its meaning and inherent structure. Such representations are important for communicating information between people, but also for building information systems which manage and exploit this information in the performance of useful tasks (Mylopoulos, 1998). A particular important activity to arrive at such representations is conceptual modeling, which is defined as "the activity of formally describing some aspects of the physical and social world around us for purposes of understanding and communication. Moreover, it supports structuring and inferential facilities that are psychologically grounded. After all, the descriptions that arise from conceptual modeling activities are intended to be used by humans, not machines". (Mylopoulos, 1992)

As any pragmatic subject, conceptual modeling can profit from *tool support* to complete its purpose. Since conceptual models are concrete artifacts, they must exist in the physical world. Therefore, in order to obtain such artifacts, users primarily need a tool for creating and editing the constructs of a modeling language. Besides editing, there is a variety of manipulations that can be done with a model (henceforth, *model manipulation*) such as analysis of its content, syntactic and semantic validation, interchange between conceptual modeling tools, persistence in different formats, generation of textual documentation, transformations to different modeling languages, etc.

Although it is possible to create conceptual models using a pen and a piece of paper (or, perhaps, cave walls and red ochre), users desire the usability and flexibility provided by computer-based tools. However, even those types of tools can be built in an *ad hoc* manner. For instance, many of the model manipulation activities can be done in different tools that do not share any implementation aspect and, as a consequence, lack integration and reusability. Such situation impacts both users and developers. On one hand, users cannot perform their modeling activities easily, since there is no integration between tools. On the other hand, developers perform extra tasks every time a new tool to support some modeling activity has to be developed, because there is no reusability.

We address these issues in this work by providing an organized infrastructure for conceptual modeling that integrates many modeling activities, such as model editing and manipulation, and promotes reusability of modeling components. More specifically, we give tool support for a well-founded conceptual modeling language, namely OntoUML (Guizzardi, 2005), providing a graphical

editor for constructing models and also a reference metamodel to be the core of model manipulation. Furthermore, we integrate these two aspects by using a transformation from the front end graphical editor's metamodel to the back end reference metamodel, hence, connecting the concrete syntax editing with the abstract syntax validation. Thus, we provide a *modeling infrastructure* for OntoUML.

1.1 BACKGROUND

Abstractions of a given portion of reality are constructed in terms of concepts, i.e., abstract representations of certain aspects of entities that exist in that portion of reality (or domain of discourse). *Conceptualization* is the set of concepts used to articulate abstractions of state of affairs in a given domain. The abstraction of a given portion of reality articulated according to a domain conceptualization is termed a *domain abstraction*. Conceptualizations and domains abstractions are abstract entities that only exist in the mind of the user or a community of users of a language. In order to be documented, communicated and analyzed, these entities must be captured in terms of some concrete artifact. The representation of a domain abstraction is named a *model*. Moreover, in order to represent a model, a *modeling language* is necessary. The relation between conceptualization, domain abstraction, modeling language and model is depicted in Figure 1.1 (terminology has been adapted here). (Guizzardi, 2005)

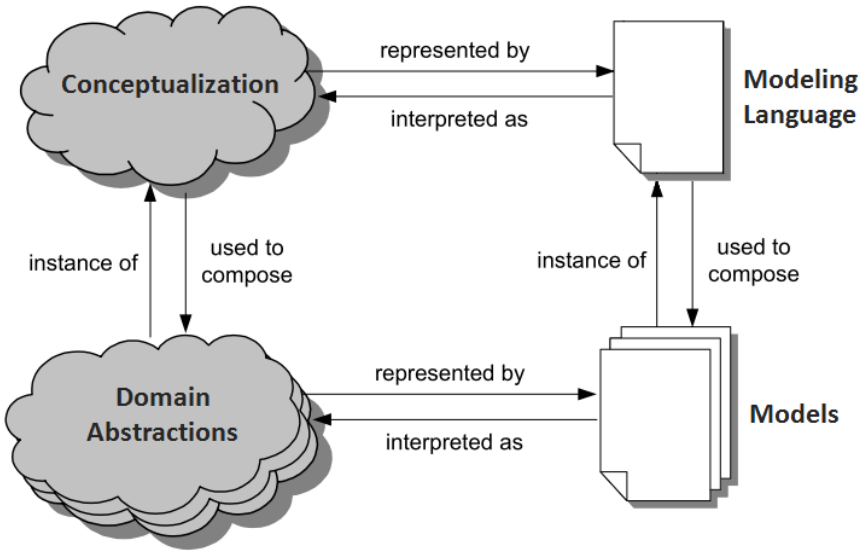


Figure 1.1 - Relation between conceptualization, domain abstraction, modeling language and model

A *foundational ontology* is a representation of theories that describe knowledge about reality in a way, which is independent of language, of particular states of affairs (states of the world), and epistemic states of knowledgeable agents. In other words, a foundational ontology is a domain-independent philosophically and cognitively well-founded system of real-world categories. It deals

with formal aspects of objects irrespective of their particular nature including concepts from theory of parts, theory of wholes, types and instantiation, identity, dependence, unity. Furthermore, a foundational ontology can be used to provide real-world semantics for general conceptual modeling languages, and to constrain the possible interpretations of their modeling primitives. (Guizzardi, 2005)

On the other hand, a *domain ontology* is a conceptual specification that describes knowledge about a *domain* (also in a manner that is independent of epistemic states and states of affairs). Instances of the concepts in a domain ontology should approximate as well as possible the phenomena in a given domain in reality. An example of a domain ontology related to genealogy is depicted in the OntoUML model of Figure 1.2.

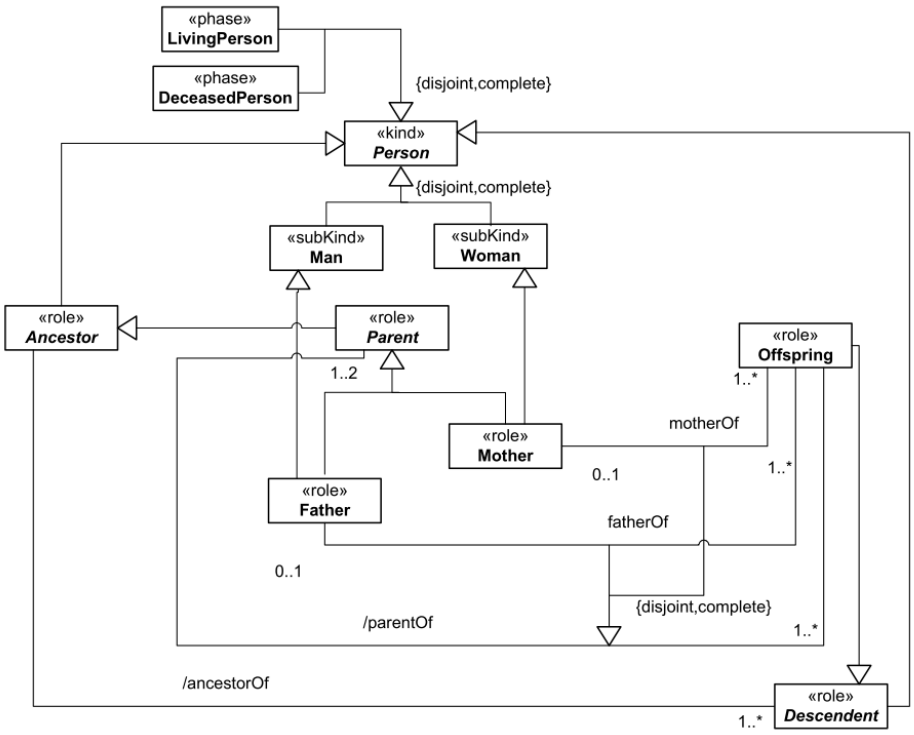


Figure 1.2 - An ontology for the genealogy domain represented in OntoUML (Guizzardi, 2005)

Now, we are able to contextualize this work in terms of the concepts of Figure 1.1. First, our *conceptualization* encompasses the concepts of a unified foundational ontology (UFO) defined in (Guizzardi, 2005). This foundational ontology is used to instantiate domain ontologies. Thus, in the context of this work, the *domain abstractions* are concepts of domain ontologies. Moreover, the concepts of UFO are applied in a UML metamodel in order to turn it into a well-founded conceptual *modeling language* named OntoUML, also defined in (Guizzardi, 2005). As a consequence, this *modeling language* is domain independent and is used to represent domain ontologies. Here, we call OntoUML *models*, like the one depicted in Figure 1.2, conceptual models. Finally, the summary of this contextualization is represented in Figure 1.3.

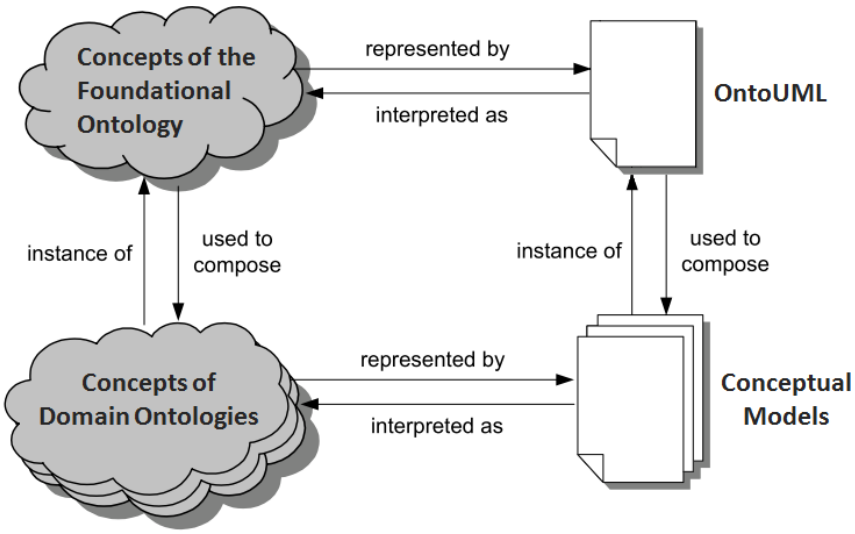


Figure 1.3 - Context of this work in terms of the concepts of Figure 1.1

1.2 Motivation and Goals

Although tool support for OntoUML conceptual models has already been developed in (Benevides, 2010), where mechanisms for building and validating the syntax of OntoUML conceptual models were provided, in our work we address some problems related to the author's approach. Basically, those problems are related to his metamodel implementation. First, it is based on a specification that is not compatible with the rules for constructing UML 2.0 Profiles, a problem whose origin traces back to the profile specification provided in (Guizzardi, 2005) before the final UML 2.0 Profile mechanism was fully developed. Secondly, it is a narrowed version of the official UML metamodel, lacking support for some language constructs. Furthermore, the platform in which the graphical editor was built forces the addition of elements that stand essentially for concrete syntax purposes into the language's metamodel, thus provoking an undesirable mixture between concrete and abstract syntax.

Consequently, one goal is to develop a metamodel that: is in agreement with the rules for forming UML Profiles (the so-called lightweight extension); uses a fully-compliant UML 2.0 metamodel implementation as a foundation; has no elements whose sole purpose is to supply concrete syntax. Then, this metamodel will be used as a *reference metamodel* for any model manipulation purposes (including abstract syntax validation).

Another goal of our work is to utilize a mechanism that supplies a graphical editor based on a UML Profile implementation. Thanks to that, we are able to put in practice our re-design of the OntoUML metamodel (this time obeying the rules for forming a UML Profile) and also provide a second alternative for users to edit models graphically. Later on, we connect our graphical editor's metamodel with the reference metamodel via a model transformation, in order to unite model

editing and syntax validation. Alternatively, we also provide a transformation from the metamodel developed in (Benevides, 2010) to our reference metamodel. This way, users can choose any of the two graphical editors and still enjoy the advantages of our reference metamodel.

By separating front-end graphical editing from back-end model manipulation, we allow the development of as many front-end tools as required and, at the same time, provide a core back-end for syntax validation and other kinds of model manipulation. As a consequence, front-end tools can implement their own metamodels with varying degrees of support for modeling elements as well as varying degrees of syntax validation. On the other hand, model transformations (such as OntoUML to Alloy (Benevides, 2010) or OntoUML to Structured English) may be defined in terms of the core reference metamodel, which is free from concrete syntax concerns and implements a full specialization of the UML 2.0 metamodel enforcing all syntactic constraints for well-formed conceptual models (as defined in (Guizzardi, 2005)). Therefore, to accomplish our tasks, we adopt a Model Driven Architecture approach (OMG, 2003) by specifying (meta)models and transformations between them.

1.3 Approach and Structure

In order to achieve our goals, we use the Eclipse software development environment[1], since it gives support to model-driven implementations. Another important aspect is that Eclipse has an extensible plug-in system that allows us to provide new functionalities on top of the runtime system. Moreover, Eclipse is free and open source software. As a consequence, we can take advantage of the original implementation of their UML 2.0 metamodel and extend it in our benefit. Finally, the Eclipse tool recently published a new mechanism for creating a graphical editor based on a UML 2.0 Profile, which will be convenient for creating our OntoUML graphical editor and benchmarking our reviewed OntoUML Profile.

With respect to the structure of this work, Chapter 2 describes the architecture of our modeling infrastructure, along with a brief description of the technologies used to implement it. In Chapter 3, we present a summary of the foundational ontology that later will be used to construct our UML Profile. In Chapter 4, we use the concepts of the foundational ontology to implement a well-defined UML 2.0 Profile in the Eclipse tool and thus obtain a graphical editor for OntoUML. Chapter 5 is focused on the implementation of the reference metamodel in the Ecore language. Chapter 6 contains the OCL syntactical constraints that were embedded in the reference metamodel. Finally, Chapter 7 discusses the conclusions and outlines some future work.

[1] http://www.eclipse.org/

2 MODELING INFRASTRUCTURE

In this chapter, we discuss the general aspects of our modeling infrastructure by describing its elements, their external behavior and the relations between them. Initially, we explain the infrastructure that existed before our work, address its problems and later we propose our new infrastructure. In addition, we provide general details about the tools used to implement the infrastructure.

2.1 PREVIOUS INFRASTRUCTURE

Before our work, tool support for OntoUML was already provided in (Benevides, 2010). Foremost, the author implemented an OntoUML graphical editor, which we call here the GMF Editor. The basic principle of an editor is that it allows the user to manipulate the concrete syntax of the language (such as boxes for classes, lines for associations, arrows for generalizations) and relate them, based on the abstract syntax specification. For instance, in UML, the user is able to connect two Classes using a Generalization (in concrete syntax terms, connect two boxes using an arrow) because the abstract syntax allows Classifiers (the supertype of Class) to participate in Generalizations. Therefore, an editor is composed of at least two components, one for defining the concrete syntax and other for the abstract syntax. The latter is also called the *metamodel* of the language. An example of the relation between these two components is exhibited in Figure 2.1 where we illustrate the infrastructure provided in Benevides' work.

Additionally, in (Benevides, 2010) and (Braga, et al., 2010), a transformation from OntoUML into formal specifications in the logic-based language Alloy (Alloy Community, 2009) was developed. This transformation was done in terms of the GMF Editor's metamodel and is represented as T_0 in Figure 2.1.

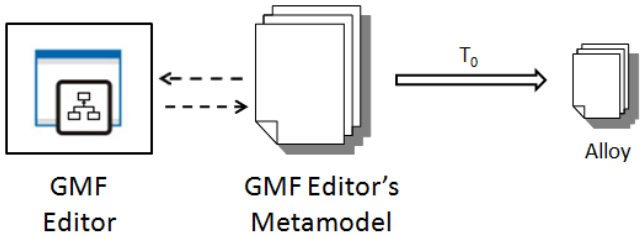


Figure 2.1 – Previous infrastructure provided in (Benevides, 2010)

In terms of implementation details, Benevides defined an OntoUML metamodel using Ecore, which is a metamodeling language that belongs to the Eclipse Modeling Framework (EMF) (The Eclipse Foundation, 2010). Then, he mapped the elements of the metamodel (abstract syntax) to concrete syntax elements (e.g., a Class is represented as a box) using the Eclipse Graphical Modeling

Framework (GMF) (The Eclipse Foundation, 2010). Notice that in Figure 2.1 we represented the dependency relation between the concrete syntax and abstract syntax in both directions. In order to fully express the OntoUML language in GMF, Benevides had to incorporate elements that support the concrete syntax in his metamodel. Thus, in this particular case, not only the concrete syntax is dependent on the abstract, as usual, but the opposite is also valid. More details about this undesirable mixture are provided in Chapter 5 of this work.

2.2 Our Infrastructure

As mentioned in Chapter 1, tools for model editing and manipulation may emerge in *ad hoc* manners. The way in which the previous infrastructure was being conducted could impact future works, because the GMF Editor's metamodel is not very qualified to be related to model manipulation in general. Primarily, it has the already addressed mixture between concrete and abstract syntax. Moreover, as we will see later, the validation aspects are mostly located in the context of the GMF editor, not on its metamodel. Finally, the metamodel implementation is not fully-compliant with the UML 2.0 metamodel, i.e., some of the constructs are not implemented in the way specified by the OMG documentation (OMG, 2009) and some of them are simply absent. More details about those issues are provided in Chapter 5, when we compare the GMF Editor's metamodel with our reference metamodel, since both are implemented in the same language. As a result, in order to promote the strict separation of concrete and abstract syntax, and solve other manipulation issues, we propose the infrastructure depicted in Figure 2.2.

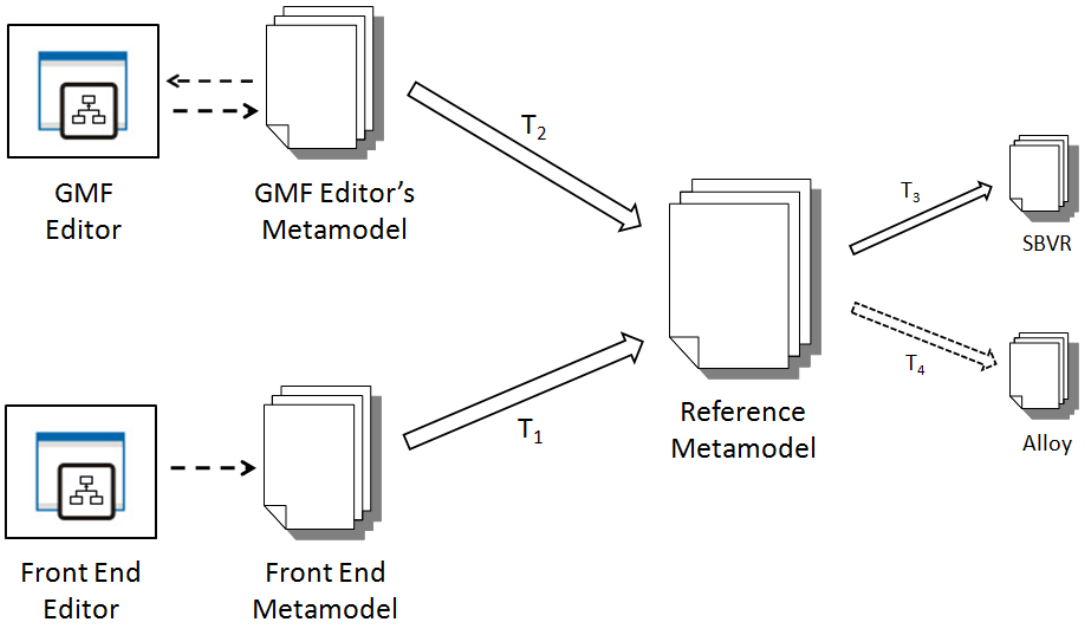


Figure 2.2 –The modeling infrastructure proposed in this work

The bottom left side of Figure 2.2 depicts the graphical editor provided in this work, which we call here the *Front End Editor*. We begin its development by defining an OntoUML metamodel using a UML Profile mechanism (OMG, 2009) (The Eclipse Foundation, 2006). In a nutshell, this profile mechanism allows the modeler to choose which UML metaclasses will be decorated with stereotypes (take for example the model of Figure 1.2; the metaclass Class is decorated with stereotypes such as kind, subKind, phase and role). This mechanism is a much easier way to implement any UML Profile, since it already provides the elements of the UML superstructure (e.g., Class, Association, Generalization). Contrariwise, in the Ecore approach (used by Benevides in his editor's metamodel and also used in our reference metamodel), one has to define from scratch every UML construct in the Ecore model. Once the metamodel is defined, the Eclipse UML Profile tool allows the user to apply the OntoUML Profile in any standard UML diagram. Therefore, the effort to obtain the front end editor is reduced to the effort of producing a UML Profile specification, since the concrete syntax mapping is already embedded in an Eclipse UML plug-in. In other words, the assignment of concrete symbols to the abstract syntax elements was already done in the implementation of standard UML editors and, therefore, does not have to be redone. Furthermore, our graphical editor's metamodel is free from concrete syntax elements, thus, it has a one way dependence, as opposed to the two way dependency of the GMF Editor (see the arrows connecting concrete and abstract syntax in both editors in Figure 2.2).

In the center of our infrastructure stands one of the main contributions of this work, namely the *reference metamodel*. This metamodel, just like Benevides' metamodel, is defined in the Ecore language. Nevertheless, it is heavily based on the official UML documentation (OMG, 2009) because we took advantage of the Eclipse implementation of the UML Superstructure in Ecore. In depth, we refined the Eclipse implementation by keeping only the convenient elements of the UML Superstructure, most of them belonging to the UML class diagram. All the other elements were removed from the Eclipse implementation and, as a consequence, we obtained a filtered UML metamodel (containing constructs such as Class, DataType, Association, Property and Package). In the sequel, we introduced in the metamodel the constructs related to OntoUML as specializations of the former UML constructs. In this process, we obeyed the rules for constructing UML Profiles in a similar way to our front end metamodel specification. For example, OntoUML constructs like Kind, Phase and Role specialize the UML metaclass Class and constructs related to Relations in general (such as part-whole, dependency relationship, formal and material relations) specialize the UML metaclass Association. The reference metamodel is specified in Chapter 5.

Another important aspect of our infrastructure is syntax validation. In both our work and Benevides' work the constraints were defined in the Object Constraint Language (OCL) (OMG, 2006), but they differ with respect to their location within the infrastructure. Benevides incorporates most

of the constraints in the GMF component of his graphical editor, not on the Ecore metamodel. Differently, we embed all the syntactical constraints defined in (Guizzardi, 2005) in our Ecore reference metamodel. Thus, the reference metamodel is the element of our infrastructure in which syntax validation independs of any model editing technology. With this approach, editors may implement constraints at various degrees (from none to all the constraints) and in various components (e.g., the metamodel or a technology-specific component like GMF) and delegate the rest of the validation to the reference metamodel through model transformation.

For instance, Benevides divides constraints in the categories of *live validation* and *batch validation*. When a user creates an invalid generalization (for example, as we will see in the forthcoming chapters, a generalization from a Kind to a Role), an editor should take the initiative and warn the constraint violation instantly. Therefore, that constraint belongs to the category of live validation. However, right after a user creates a Role, the model is currently incomplete (since a Role must be connected to a mediation, as we will see later), but such constraint violation should not be warned immediately. In this case, the user has to take the initiative and request a validation in the model, so such constraint violation can be warned. Hence, that constraint belongs to the category of batch validation. Based on this taxonomy, an editor could implement live validation constraints and take benefit from the reference metamodel for evaluating the batch validation constraints[2].

In order to integrate the front end metamodels underlying the editors with the back end reference metamodel, we provided the transformations T_1 and T_2 depicted in Figure 2.2. In depth, EMF supplies an automatic Java code generation based on an Ecore model. That way, every construct of the Ecore model is mapped to some Java construct. For instance, EClasses become Java classes and EReferences become attributes in Java classes (EClass and EReference are defined in Chapter 5). Even our front end metamodel has an Ecore foundation, since it is implicitly connected to the Eclipse implementation of the UML Superstructure (the same implementation that was the foundation for our reference metamodel). Hence, all the three metamodels have an Ecore foundation and have Java classes related to it. As a consequence, both transformations T_1 and T_2 are implemented in Java and map the instances of Java classes of the source metamodel to instances of Java classes of the target metamodel. Later, the instances of the target Java classes are serialized back into an Ecore model, thus, providing an instance of the reference metamodel as a final product of T_1 and T_2. In other words, T_1 takes as an argument an *instance* of our front end metamodel (i.e., an OntoUML model written as a stereotyped class diagram) and transforms it into an *instance* of the reference

[2] Nothing forbids an editor to perform live validation with the aid of the reference metamodel. For instance, the user makes an action, the editor converts the model to the reference metamodel syntax, performs the validation in the reference metamodel, converts the model back to its former syntax and, finally, presents the warning to the user.

metamodel (which is the same OntoUML model but, this time, written in terms of a different abstract syntax).

As a consequence, an editor's metamodel could partially implement the constructs of the UML specification (e.g., abstain from implementing the concept of Packages, like Benevides does) and, afterwards, models written in the editor could be transformed into instances of the reference metamodel, following the same pattern of T_1 and T_2.

Finally, the right part of Figure 2.2 demonstrates that model transformations relating OntoUML to different languages (as opposed to T_1 and T_2 that transform OntoUML into OntoUML itself) may occur using the reference metamodel, irrespective of the front end metamodels. For instance, in another work, we transformed OntoUML into Semantics of Business Vocabulary and Business Rules (SBVR) (OMG, 2008) using our reference metamodel as a source. This transformation is represented in the infrastructure as T_3. With our modeling infrastructure, the transformation from OntoUML to Alloy would be done in terms of the reference metamodel, as represented in T_4.

In summary, even if new technologies arise and a particular editor becomes obsolete, new developments will not compromise any external model transformations (i.e., relating OntoUML to different languages) or other kinds of model manipulation (such as syntax validation). This happens because model manipulations are done in terms of the reference metamodel and not in terms of any specific editor's metamodel or implementation.

3 FOUNDATIONAL ONTOLOGY

This chapter is heavily based on (Guizzardi, 2005), therefore, quotation marks will be omitted. There, the author provides an ontological foundation for the most fundamental concepts in conceptual modeling. These foundations comprise a number of ontological theories which are built on established work on philosophical ontology, cognitive psychology, philosophy of language and linguistics. Together, these theories amount a system of categories and formal relations known as a foundational ontology.

Once constructed, this foundational ontology is used as a reference model prescribing the concepts that should be countenanced by a well-founded conceptual modeling language and providing real-world semantics for the language constructs representing these concepts. Also, the reference ontology proposed is focused on providing foundations for the most fundamental and widespread constructs for conceptual modeling languages, namely, types and types taxonomies, roles, attributes, attribute values and attribute value spaces, relationships and part-whole relations.

Besides the theoretical work, the approach defended on Guizzardi's thesis was instantiated by proposing a concrete conceptual modeling language that incorporates the foundations captured in the reference ontology. Therefore, a specific modeling language, namely, the Unified Modeling Language (UML) (OMG, 2003c) was analyzed and redesigned with the objective of proposing an ontologically well-founded version of it that can be used as an appropriate conceptual modeling language. This proposed extension of UML was later called OntoUML.

In the sequel, we gradually exhibit some important concepts of the foundational ontology, which is part of a larger one called Unified Foundational Ontology (UFO). Such concepts set the basis for the construction of the UML Profile in Chapter 4.

3.1 SUBSTANTIALS

The first distinction to be made is between Individuals and Universals. *Universals*, i.e., space-time independent pattern of features that can be realized in a number of *Individuals* (instances). For example, in some situation, Mary is an individual (instance) of the universal Woman, while John is an individual (instance) of the universal Child, and both are instances of the universal Person. Thus, we define the semantics of the subtype relation between Universals using the notion of possible worlds.

Definition (subtype): Relates a specific Classifier C_1 to a general Classifier C_2. In every world w in which x is an instance of C_1, x is also an instance of C_2.

Additionally, in this section we are interested in a specific type of individuals called *Substantials*, i.e., entities that persist in time while keeping their identity (as opposed to *events* such as a kiss, a business process or a birthday party). Examples include physical and social persisting

entities of everyday experience such as balls, rocks, students, the North Sea and Queen Beatrix. Therefore, Person and its subtypes (viz. Man, Woman, Child, Teenager, Adult, Student, etc) are *Substantial Universals*, i.e., Universals whose instances are Substantials.

A principle of application is that in accordance with which we judge whether a general term applies to a particular (e.g., whether something is a Person, a Dog, a Chair or a Student). A principle of identity supports the judgment whether two particulars are the same, i.e., in which circumstances the identity relation holds. Therefore, Substantial Universals that supply a principle of application and principle of identity for the individuals they collect are called *Sortal Universals*. The next specialization of Sortal Universals is based on an important meta-property called *rigidity*.

Definition (Rigidity): A type T is rigid if for every instance x of T, x is necessarily (in the modal sense) an instance of T. In other words, if x instantiates T in a given world w, then x must instantiate T in every world w'.

Definition (Anti-Rigidity): A type T is anti-rigid if for every instance x of T, x is possibly (in the modal sense) not an instance of T. In other words, if x instantiates T in a given world w, then there is a possible world w' in which x does not instantiate T.

An example of rigid universal would be Person, since instances of Person cannot cease to be so without ceasing to exist. Conversely, Child is an example of anti-rigid universal, since for every instance of Child it is possible to imagine a world in which it ceases to be so but continues to exist, as a Teenager or an Adult, for instance. Analogously, Man and Woman are rigid universals and on the other hand Student, Teenager and Adult are anti-rigid universals.

A rigid sortal can be a Substance Sortal or a SubKind. A Substance Sortal is the unique sortal that provides an identity principle to its instances (e.g., Person, Sand and Forest). Substance Sortals can be specialized in other rigid subtypes that inherit their supplied principle of identity, named *SubKinds* (e.g., Woman and Man, which inherits its identity principle from Person). Consider, for example, the individual Mick Jagger both now (when he is the lead singer of Rolling Stones and 66 years old), in the past (when he was the boy Mike Philip living in Kent, England) or in a counterfactual situation (such as the one in which he decided to continue in the London School of Economics and has never pursued a musical career). We would like to say that the boy Mike Philip is identical to the man Mick Jagger that he later became. Hence, the sortal Person is the unique substance sortal that defines the validity of that claim, that is, Mike Philip persists through changes in height, weight, age, residence, etc, as the same individual. Furthermore, there are three types of Substance Sortals, namely Kind, Quantity and Collective. Quantities are nominalizations of amounts of matter like water, sand, sugar, martini and wine, and Collectives are representations of collections in general, such as forest, deck of cards, pile of bricks and pack of wolves. Both terms will be formally defined

when we discuss parts and wholes. Also, if a Substance Sortal does not fall into these two categories, it is named a *Kind*.

On the other hand, anti-rigid sortals are specialized into Phases and Roles. *Phases* constitute possible stages in the history of a substance sortal. Examples include: Alive and Deceased as possible stages of a Person; Caterpillar and Butterfly of a Lepidopteran; Town and Metropolis of a City; Boy, Male Teenager and Adult Male of a Male Person. Universals representing Phases constitute a partition of the Substance Sortal they specialize. For example, if <Alive, Deceased> is a phase-partition of a substance sortal Person then for every world w, every Person x is either an instance of Alive or of Deceased but not of both. Moreover, if x is an instance of Alive in world w then there is a world w' such that x is not an instance of Alive in w', which in this case, implies that x is an instance of Deceased in w'.

Contrary to Phases, *Roles* do not necessarily form a partition of substance sortals. Moreover, they differ from Phases with respect to the specialization condition. For a Phase P, it depends solely on intrinsic properties of P. For instance, one might say that if Mick Jagger is a Living Person then he is a Person who has the property of being alive or, if Spot is a Puppy then it is a Dog who has the property of being less than a year old. For a Role R, conversely, it depends on extrinsic (relational) properties of R. For example, one might say that if John is a Student then John is a Person who is enrolled in some educational institution or that, if Peter is a Customer then Peter is a Person who buys a Product y from a Supplier z. In other words, an entity plays a role in a certain context, demarcated by its relation with other entities.

The other type of Substantial Universals, complementary to Sortal Universal, is the Mixin Universal. *Mixins* represent an abstraction of properties that are common to multiple disjoint types. For example, the Mixin Rational Entity can be judged to represent an essential property that is common to all its instances and it is itself a rigid type. A Rigid Mixin is called *Category*. In contrast, some Mixins are anti-rigid and represent abstractions of common properties of roles. These types are called *RoleMixins* and represent any abstraction of common contingent properties of multiple disjoint roles. An example of RoleMixin would be Customer, generalizing common properties of the Role Personal Customer (which is a Person) and the Role Corporate Customer (which is an Organization). Moreover, some Mixins represent properties that are essential to some of its instances and accidental to others. This meta-property is called non-rigidity, a weaker constraint than anti-rigidity. An example is the Mixin Seatable, which represents a property that can be considered essential to the Kinds Chair and Stool but accidental to Crate, Paper Box or Rock. This type of Mixins are just called *Mixins* without further qualification.

Finally, we present the so-called typology of substantial universals in Figure 3.1, which is a compilation of the theory presented so far.

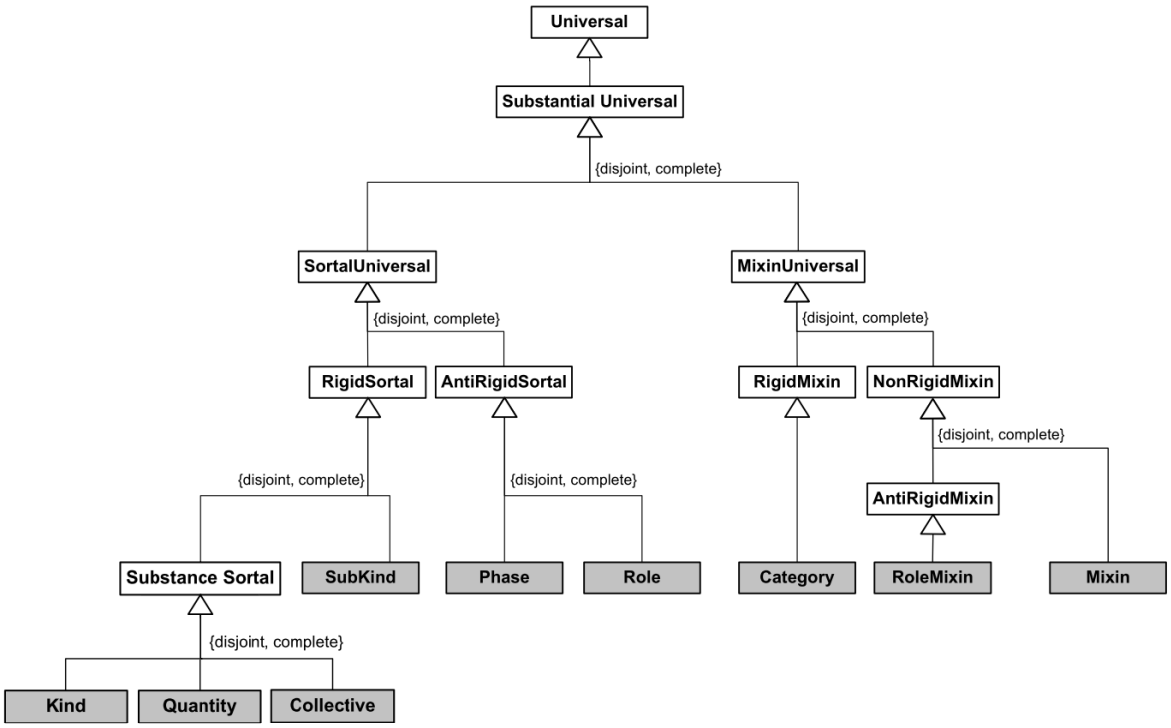


Figure 3.1 - Ontological distinctions in a Typology of Substantial Universals

3.2 FOUR-CATEGORY ONTOLOGY

In the previous section, we presented the distinction between Individuals and Universals and also many details about Substantial Universals. In this section, we shall present some categories that were previously omitted in our discourse.

As discussed in (Guizzardi, 2005), a bicategorial ontology based on sets and its members (as supplied by set theory) does not constitute a suitable inventory of formal entities that can be used to model reality. Thus, a fundamental distinction in this ontology is between the categories of the so-called *urelements* and *sets*. Urelements are entities that are not sets. They form an ultimate layer of entities without any set-theoretical structure in their build-up, i.e., neither the membership relation nor the set inclusion relation can unfold the internal structure of urelements. And, the most important, urelements are classified into two (already mentioned) main categories: *individuals* and *universals*. Next, we make a refinement in the category of individuals.

Traditionally, in the philosophical literature, there is a fundamental distinction in the category of individuals between enduring and perduring entities (henceforth, named endurants and perdurants). Classically, the distinction between them can be understood in terms of their behavior with respect to time. Endurants are said to be wholly present whenever they are present, i.e., they *are in time*, in the sense that if we say that in circumstance c_1 an endurant *e* has a property P_1 and in circumstance c_2 the property P_2 (possibly incompatible with P_1), it is the very same endurant *e* that we refer to in each of these situations. Examples of endurants are a house, a person, the moon, a

hole, an amount of sand. For instance, we can say that an individual John weights 80kg at c_1 but 68kg at c_2. Nonetheless, we are in these two cases referring to the same individual, namely the person John. Perdurants are individuals composed of temporal parts, they *happen in time* in the sense that they extend in time accumulating temporal parts. Examples of perdurants are a race, a conversation, a football game, a symphony execution, a birthday party, the Second World War and a business process.

The ontology proposed in (Guizzardi, 2005) accounts for a descriptive commonsensical view of reality, focused on structural (as opposed to dynamic) aspects. For this reason, the foundational ontology is an *ontology of endurants*. Therefore, our discussion is focused on endurant individuals and endurant universals. Finally, we are able to show the core of the urelement fragment of the ontology in Figure 3.2, which amounts to a so-called *Four-category ontology*. The categories comprising this ontology are two pairs individuals-universals, namely substantial and substantial universals; moments and moment universals. The first pair was already discussed on a previous section while the second one will be discussed in the sequel.

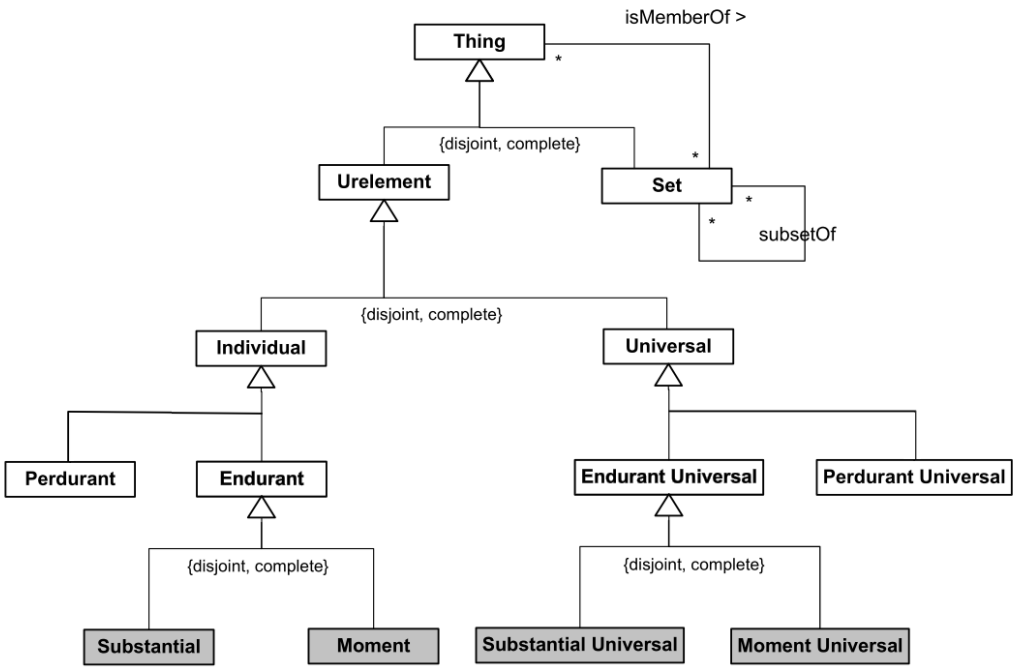


Figure 3.2 - The urelement fragment of the foundational ontology centered in a four-categorical account

3.3 MOMENTS

The notion of moment employed here comprises Intrinsic Moments and Relational Moments (or relators). Examples of the former are *qualities* such as a color, a weight, an electric charge, a circular shape; *modes* such as a thought, a skill, a belief, an intention, a headache, as well as dispositions such as the refrangibility property of light rays, or the disposition of a magnetic material to attract a

metallic object. Examples of the latter are a kiss, a handshake, a covalent bond, but also social objects such as a flight connection, a purchase order and a commitment or claim.

An important feature that characterizes all *moments* is that they can only exist in other individuals (in the way in which, for example, electrical charge can only exist in some conductor). To put it more technically, we say that moments are *existentially dependent* on other individuals, named their bearer.

Definition (Existential Dependence): An individual *x* is existentially dependent on another individual *y* iff, as a matter of necessity, *y* must exist whenever *x* exists.

Existential dependence is a necessary but not a sufficient condition for something to be a moment. For instance, the temperature of a volume of a gas depends on, but is not a moment of its pressure. Thus, for an individual *x* to be a moment of another individual *y* (its bearer), a relation of *inherence* must hold between the two. For example, inherence glues your smile to your face, or the charge in a specific conductor to the conductor itself. Therefore, inherence is a special type of existential dependence relation between particulars. Also, inherence is an irreflexive, asymmetric and intransitive relation between moments and other types of endurants.

Consequently, we define a moment as an endurant that inheres in another endurant and, on the other hand, a substantial as an endurant that does not inhere in another endurant, i.e., which is not a moment.

3.3.1 QUALITIES

For several perceivable or conceivable moment universals there is an associated *quality dimension* in human cognition. For example, height and mass are associated with one-dimensional structures with a zero point isomorphic to the half-line of nonnegative numbers. Other moments such as color and taste are represented by several dimensions. For instance, taste can be represented as a tetrahedron space comprising the dimensions of saline, sweet, bitter and sour, and color can be represented in terms of the dimensions of hue, saturation and brightness.

Certain quality dimensions are *integral* in the sense that one cannot assign an object value on one dimension without giving the value on the other. For example, an object cannot be given a hue without giving it a brightness value. Dimensions that are not integral are said to be *separable*, as for example the size and hue dimensions. Therefore, *quality domain* is defined as a set of integral dimensions that are separable from all other dimensions. Additionally, the perception or conception of a moment individual can be represented as a point in a quality domain, namely *quale*.

We adopt the term *quality structures* to refer to quality dimensions and quality domains. Furthermore, we use the term *quality universals* for those intrinsic moment universals that are associated with a quality structure. Finally, we name *quality* a moment individual that instantiates a quality universal. For the same quality universal, there can be potentially many collections of quality

domains associated with it, and which one is to be adopted depends on the objectives underlying each specific conceptualization.

To materialize this theory, consider this example: suppose we have two distinct particular substantials *a* (a red apple) and *b* (a red car), and two qualities q_1 (particular color of *a*) and q_2 (particular color of *b*). When saying that *a* and *b* have the same color, we mean that their individual color qualities q_1 and q_2 are (numerically) different, however, they can both be mapped to the same point in the color quality domain, i.e., they have the same quale. The relations between a substantial, one of its qualities and the associated quale are summarized in Figure 3.3.

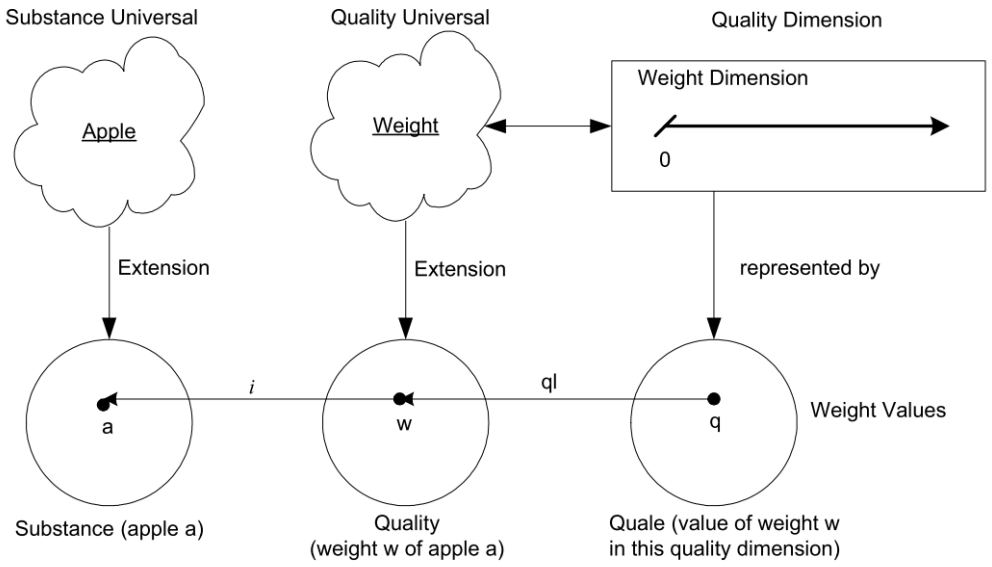


Figure 3.3 - Substantial, Quality and Quale

3.3.2 Relations and Relators

Relations are entities that glue together other entities. Every relation has a number of relata as arguments, which are connected or related by it. The number of a relation's arguments is called arity. As much as an unary property such as *being red*, properties of higher arities such as *being married-to*, *being heavier-than* are universals, since they can be predicated on a multitude of individuals.

We divide relations into two broad categories, called material and formal relations. *Formal relations* hold between two or more entities directly without any further intervening individual. Examples of formal relations are: 5 *is greater than* 3, this day *is part of* this month, and N *is subset of* Q, but also the relations of *inherence*, *existential dependence*, *instantiation*, among others. We also classify as formal those relations of comparison such as *is taller than*, *is older than*, *knows more Greek than*. The entities that are immediate relata of such relations are not substantials but qualities. To understand this, we quote Mulligan & Smith (Mulligan, et al., 1986): "One thereby discovers that, at least in the majority of cases, it is not substances but rather individual states – and dependent

entities in general – which are the immediate relata of such relations. Thus 'Hans is hotter than Erna' is made true, we may suppose, by a relation of *difference* between the individual moments which are heat-states (of differing intensities) inhering, respectively, in Hans and Erna. And then the material content of *is hotter than* is as it were distributed between the two relata". In the sequel they mention "Once this distribution has been effected, the two relata are seen to fall apart, in such a way that they no longer have anything specifically to do *with each other* but can serve equally as terms in a potentially infinite number of comparisons".

Material relations, conversely, have material structure on their own and include examples such as employments, kisses, enrollments, flight connections and commitments. The relata of a material relation are mediated by individuals that are called *relators*. Relators are individuals with the power of connecting entities; a flight connection, for example, founds a relator that connects airports, an enrollment is a relator that connects a student with an educational institution. Quoting once again Mulligan & Smith (ibid.): "The relata of real material relations such as hittings and kissings, in contrast, cannot be made to fall apart in this way: Erna's hitting, *r*, is a hitting *of Hans*; it is not a hitting of anyone and everyone who happens to play a role as a patient of a hitting qualitatively identical with *r*. Hence the relational core of such relations cannot be shown to be merely formal".

Also, one may see the difference between formal and material relations in the following perspective: while the latter alters the history of the involved relata, the former does not. For instance, the individual histories of John and Mary are different because of the relation "John *kisses* Mary" whilst the same is not true for the relation "John *is taller than* Mary". Perhaps a stronger and more general way to characterize the difference between formal and material relations is based on their foundation. However, before we proceed, there are some important notions that must be defined.

There are moments that, as much as substantial universals, can be conceptualized in terms of multiple separable quality dimensions. Examples include beliefs, desires, intentions, perceptions, symptoms, skills, among many others. We term these entities *modes*. Like substantials, modes can bear other moments, and each of these moments can be qualities referring to separable quality dimensions. However, since they are moments, differently from substantials, modes inhere necessarily in some bearer. Thus, we define a mode as an intrinsic moment individual which is not a quality.

A special type of mode that is of interest here is the so-called *externally dependent modes*. They are individual modes that inhere in a single individual but that are existentially dependent on (possibly a multitude of) other individuals that are independent of their bearers. Take, for instance, *being the father of*. This is an example of a universal property, since it is clearly multiple instantiated. Suppose that John is the father of Paul. According to our view of universals, in this case, there is a

particular instance *x* of *being the father of* which bears relations of existential dependence to both John and Paul. However, *x* is not equally dependent on the two individuals, since x is a moment it must inhere in some individual, in this case, John.

Now, suppose that John is married to Mary. There are many externally dependent modes of John that depend on the existence of Mary, and that have the same foundation (e.g., a wedding event or a social contract between parts). These are, for example, all responsibilities that John acquires by virtue of this foundation. As a consequence, we can define an individual that bears all externally dependent modes of John that share the same dependencies and the same foundation. We term this particular a *qua individual*. Qua individuals are, thus, treated here as a special type of *complex externally dependent modes*. Intuitively, a qua individual is the way an object participates in a certain relation and the name comes from considering an individual only w.r.t. certain aspects (e.g., John qua student; Mary qua musician).

Finally, we can define an aggregate of all qua individuals that share the same foundation, and name this individual a *relator*. Now, let *x*, *y* and *z* be three distinct individuals such that: (a) *x* is a relator; (b) *y* is a qua individual and *y* is part of *x*; (c) *y* inheres in *z*. In this case, we say that *x mediates z*.

We require that a relator mediates at least two distinct individuals. Also, a relator is considered here as a special type of *moment*. Thus, a relator must inhere in a unique individual, i.e., it must have a bearer. We therefore estipulate that the bearer of a relator is the (mereological) sum of the individuals that it mediates.

A relator universal is a universal whose instances are relators. We define *mediation* as a relation that holds between a universal U and a relator universal U_R iff every instance of U is mediated by an instance of U_R. As a result, relator universals constitute the basis for defining material relations R whose instances are n-tuples of entities. A relation R is called *material* if, for every n-tuple (instance of the material relation), there is a relator (instance of a relator universal U_R) that mediates every individual of the n-tuple. In this case, we say that the relation R *is derived from* the relator universal U_R, this is the so-called *derivation* relation. Otherwise, if such a relator universal U_R does not exists, R is termed a *formal* relation.

An example of ternary material relation is *purchase-from* corresponding to a relator universal *Purchase* whose instances are individual purchases. These individual purchases connect three individuals: a *person*, say *John*; an individual *good*, e.g. the book *Speech Acts By Searle*; a *shop*, say *Amazon*. Also, there is a relator *p*, instance of *Purchase*, that mediates the individuals *John*, *Speech Acts By Searle* and *Amazon*. The summary of the entities of this section and their interrelationships is depicted in Figure 3.4.

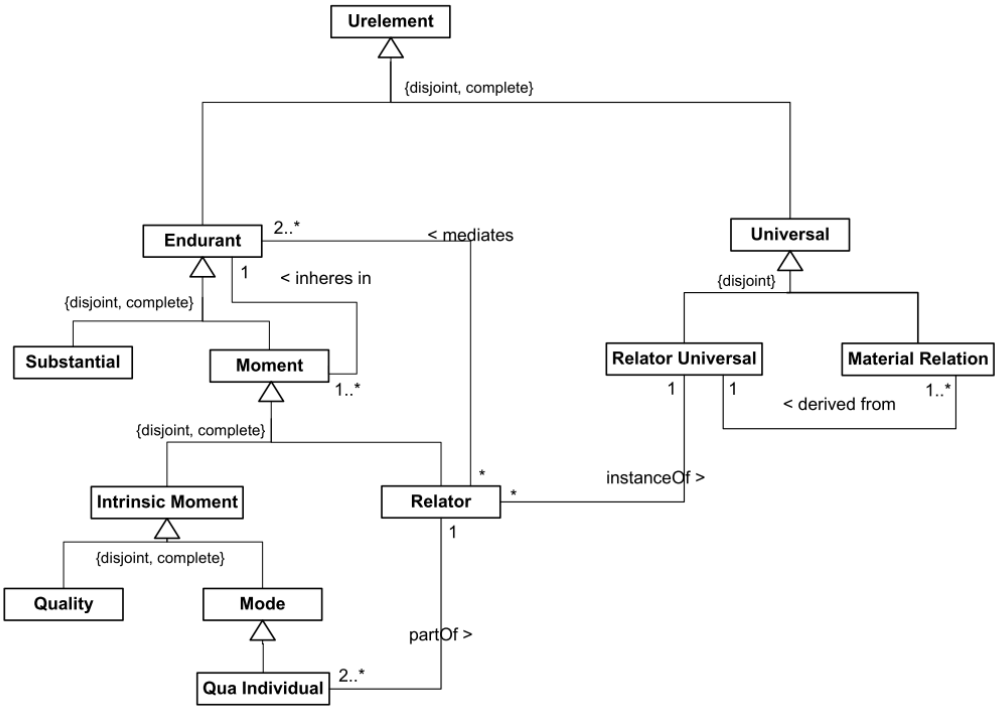


Figure 3.4 - Modes, Qua Individuals, Relators and Material Relations

3.4 Parts and Wholes

Parthood is a relation of significant importance in conceptual modeling, being present in practically all conceptual/object-oriented modeling languages. Nonetheless, in many of those languages, the concepts of part and whole are understood only intuitively, or are based on the very minimal axiomatization that these notions require. For this reason, a theory of parts and wholes is considered as a fundamental part of the foundational ontology.

3.4.1 Secondary Properties of Part-Whole Relations

In this section, we discuss secondary characteristics of part-whole relations, in the sense that it is not the case that they are properties held by all relations of this type. In contrast, they can be used to differentiate distinct types of parthood.

We begin our discussion with the concept of *shareability*. Consider the situation in which a person, say John Smith, is both a researcher and a family member and, therefore, is part of both a research group and a family (see Figure 3.5). In order to model that John belongs to a single family but at the same time possibly many research groups, we need the following definition of *non-shareability*.

Definition (non-shareable part): An individual *x* of type A is said to be a non-shareable part of another individual *y* of type B iff *y* is the only B that has *x* as part.

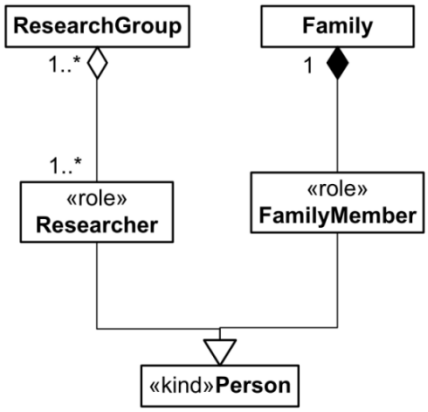


Figure 3.5 - Examples of shareable (hollow diamond) and non-shareable (black diamond) parthood relations

Now, we define the concept of *generic dependence* which is similar to the previously defined concept of *existential dependence*, but relates an individual and an universal instead of two individuals. In the context of part-whole relations, we may say *specific dependence* when referring to the case of existential dependence.

Definition (generic dependence): An individual *y* is generic dependent on a universal U iff, whenever *y* exists, it is necessarily that an instance of U exists.

In part-whole relations, (generic and specific) dependence may occur in both directions, i.e., the part may be dependent on the whole and the whole may be dependent on the part. First, we consider the latter case in Figure 3.6 and Figure 3.7. In Figure 3.6a, every person has a brain as part, and in every world that the person exists, the very same brain exists and is a part of that person. In Figure 3.6b, we have an analogous example: a car depends specifically on its chassis, thus, the part-whole relation between car and chassis holds in every world that the car exists. To put it in a different way, if the chassis is removed, the car ceases to exist as such, i.e., it loses its identity. When a whole *depends specifically* on a part, that part is called an *essential part*.

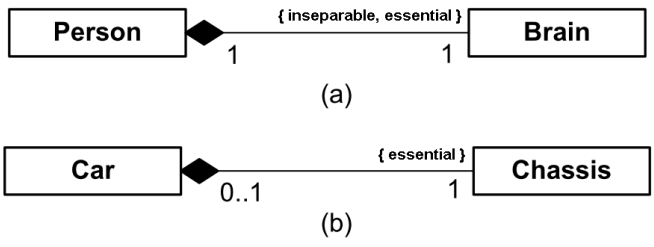


Figure 3.6 - Wholes depending specifically on their parts

Typically, the relation between a person and his/her brain is not of the same nature as the relation between a person and his/her heart. Differently from the former, it is not the case that a person depends on a *specific* heart. For instance, the fact that an individual John had the same heart during

his entire lifetime was only accidental. With the advent of heart transplants, one can easily imagine a counterfactual in which John had been transplanted a different heart. This situation is represented in Figure 3.7a. An analogous argument can be made in the case of Figure 3.7b. Although every car needs an engine, it certainly does not have to be the same engine in every possible world. When a whole *depends generically* on a part, that part is called a *mandatory part*.

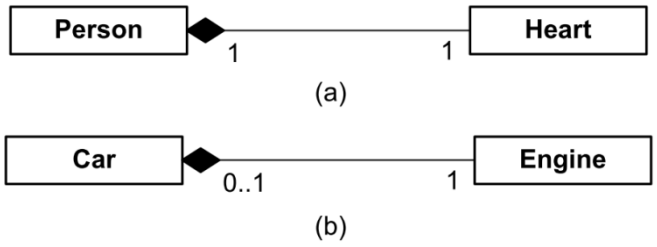


Figure 3.7 - Wholes depending generically on their parts

Similarly, a part may be dependent on its whole. In the case of Figure 3.6a, a brain also *depends specifically* on its host person, we name that an *inseparable part*. On the other hand, a heart in Figure 3.7a must be part of *a* person, only not necessarily the same person in all possible circumstances. For these cases of *generic dependence* from the part to the whole, we use the term *mandatory whole*.

3.4.2 QUANTITIES

Amounts of matter, such as water, sand, sugar, martini and wine, lack both individuation and counting principle. One cannot say, for example, “exactly five water” since arbitrarily many parts of water are still water. In order to be able to make viable references to general terms which are not count nouns, they first must be nominalized. A nominalization of a mass noun promotes the shift to the category of count nouns (e.g., a lump of clay, the water in the bathtub, a cube of sugar) and allows the representation of the corresponding sortal universals. Therefore, Guizzardi proposed a pattern for representing the so-called quantities.

A *quantity* is a maximally self-connected portion which is generically dependent on the container it constitutes. In this sense, a quantity is an instance of a genuine substance sortal universal, i.e., it has definite individuation, identity and counting principles. However, it can still be composed of other quantities of different types, e.g., wine is composed of alcohol and martini is composed of gin. All the parts of a quantity, i.e., the other quantities from which it is composed of, are essential. Also, by saying that the dependence relation between a quantity and its container is generic, it means that for the same maximally self-connected quantity there can be several “container phases”. This idea is represented in the model of Figure 3.8. A vintage is an object constituted by (possibly many) quantities of wine, however it is not a quantity, since it can be scattered over many quantities. Moreover, it is not necessary for its constituent quantities to be

essential: even if the quantity of wine now stored in a certain tank is destroyed, we still have numerically the same vintage.

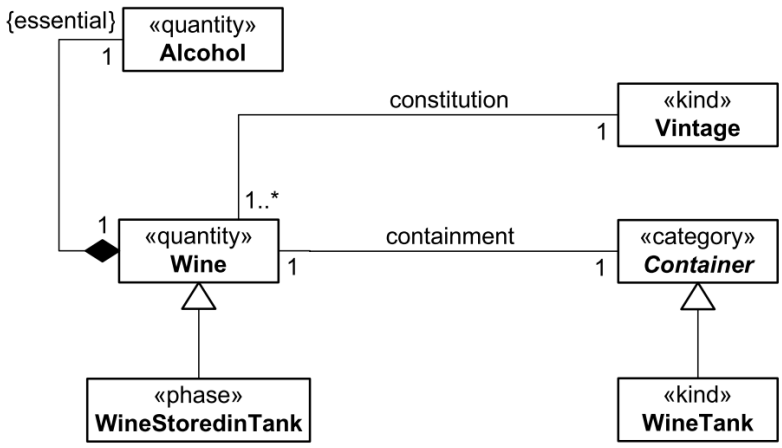


Figure 3.8 - A pattern for representing Quantities

3.4.3 COLLECTIONS

Collections are another type of substance sortal that form a complete partition along with Kinds and Quantities. Examples include tree-forest, card-deck, brick-pile, wolf-pack. Collections have parts that are not of the same kind (e.g., a tree is not a forest) and they are not infinitely divisible.

A common pattern to model collections is to consider groups as maximal sums. Consider, for instance, a group of people composed by all those people that are attending a certain museum exhibition at a certain time. There can be no group in this same sense that is part of another group of people. Nonetheless, it can be the case that, among the parts of a group of people, further structure is obtained by the presence of other collections unified by different relations. For example, it can be the case that among the parts of a group of people A, there are collections B and C composed of the English and Dutch speakers, respectively (see Figure 3.9). Neither the English speaking segment nor the Dutch speaking segment is a group of people in the technical sense just defined, since a group of people has properties that apply to none of them (e.g., the property of having both English and Dutch segments). Moreover, the unifying relations of B and C are both specializations of A's unifying relation. For example, A is the collection of all parties attending an exhibition and B is the collection of all English speakers among the parties attending the same exhibition. For this reason, transitivity holds unrestricted among the part-whole relations depicted in Figure 3.9.

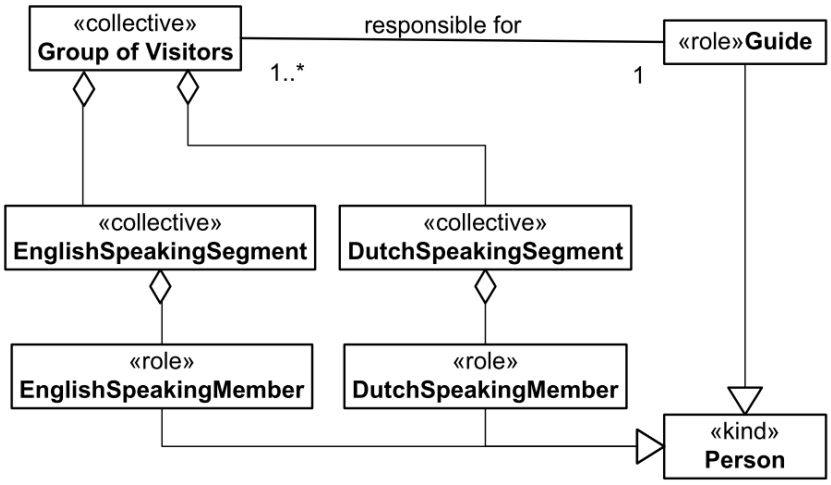


Figure 3.9 – A pattern for representing Collections

In a collection, all member parts play the same role type. For example, all trees in a forest can be said to play the role of a forest member. However, a tree is not necessarily a forest-part, i.e., the latter is an anti-rigid concept, and representing the part-whole relation between forests and trees via the role type *forest-tree* prevents one from having to specify minimum cardinality constraints which are zero. In functional complexes (see next subsection), conversely, a variety of roles can be played by different components. For example, if all ships of a fleet are conceptualized as playing solely the role of "member of a fleet" then it can be said to be a collection. Contrariwise, if this role is further specialized in "leading ship", "defense ship", "storage ship" and so forth, the fleet must be conceived as a functional complex.

Finally, we emphasize that, different from *quantities*, collectives do not necessarily obey an extensional principle of identity. Some collectives can be considered extensional by certain conceptualizations. In this case, the addition or subtraction of a member renders a different collective. However, we also consider here the existence of intentional collectives obeying non-extensional principles of identity.

3.4.4 Types of Part-Whole Relation

In this subsection, we discuss the different types of part-whole relations, namely subQuantityOf, memberOf, componentOf and subCollectionOf, and how they are related with the meta-properties covered in the previous subsections.

The *subQuantityOf* has the following meta-properties: only holds between quantities; is non-shareable; the part is essential to the whole; the cardinality constraints in both association ends are of one and exactly one; it is transitive, i.e., for all a, b, c, if Q(a,b) and Q(b,c) then Q(a,c). In the example depicted in Figure 3.10, we conclude that 'C is part of A' as a result of the transitivity between 'C being part of B' and 'B being part of A'. Moreover, B can only be part of one single

quantity A, since A is a maximal portion, and A has at maximum one quantity of B as part, since B is also a maximal portion.

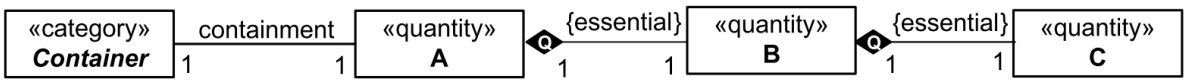


Figure 3.10 - Part-Whole relations among quantities

The *memberOf* relation is the relation between, for example, John and a group of men, the cow Joanne and the herd or the province of Overijssel and The Netherlands. MemberOf relations are never transitive, i.e., for all a, b, c, if M(a,b) and M(b,c) then ¬M(a,c). Consider the example "I am member of a club (collection) and my club is a member of an International body (collection). However, it does not follow that I am a member of this International body since this only has clubs as members, not individuals". This example is depicted in Figure 3.11.

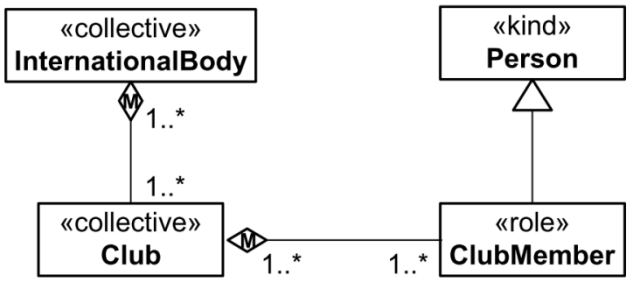


Figure 3.11 - Examples of memberOf part-whole relation

The *subCollectionOf* relation, conversely, is a relation that holds between two collectives, such that all atoms of the first are also atoms of the second. Subcollections are always created by refining the unifying relation of a certain collection. In the example of Figure 3.9, the *English Speaking Segment* is part of the *Group of Visitors*. The latter is unified by, for instance, taking all people that visit an exhibition x at date y. The former by taking all people that visit an exhibition x at date y, and that speak English. As another example, take a forest and the north part of the forest. The latter is a part of (subcollection of) the former. The forest is unified by taking all the trees located in a given area A. The north part of the forest is unified by taking all the trees located in the north part of A.

The subCollectionOf has the following meta-properties: the part is essential to the whole; the cardinality constraints in the association end relative to the part is one and exactly one; only holds between collectives; it is transitive, i.e., for all a, b, c, if C(a,b) and C(b,c) then C(a,c). In Figure 3.12, this transitivity implies that every C is part of A.

Notice that if we have M(x,y) and C(y,z) then it is also the case that M(x,z). Therefore, in Figure 3.12 we have that every D is a part of (member of) a B and also a member of A. In Figure 3.9, this means that every *English Speaking Member* is part of (member of) the *Group of Visitors*.

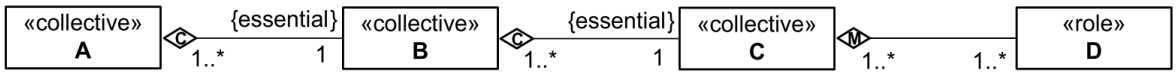


Figure 3.12 - subCollectionOf and memberOf part-whole relations

Finally, we turn our attention to the last and most important type of parthood relation for the purposes of conceptual modeling, namely *componentOf*. Like mentioned before, differently from collectives, functional complexes are composed by parts that play a multitude of roles in the context of wholes. The parts of a complex have in common that they all possess a functional link with the complex. In other words, they all contribute to the functionality (or the behavior) of the complex. For all complexes, if x is an essential part of y then y is *functionally dependent* on x.

Consider the example depicted in Figure 3.13. By the definition of essential parthood, a Person is existentially dependent on his/her (specific) Brain and a Brain is existentially dependent on its (specific) Cerebellum. Thus, we conclude that every Person is dependent on a specific Cerebellum. If the Cerebellum x is not part of the Person y, we have to conclude that y is existentially dependent on a substantial that is disjoint from it and, consequently, that particular person y is not a substantial. This result is certainly absurd. Therefore, we must conclude that transitivity always holds across essential parthood relations.

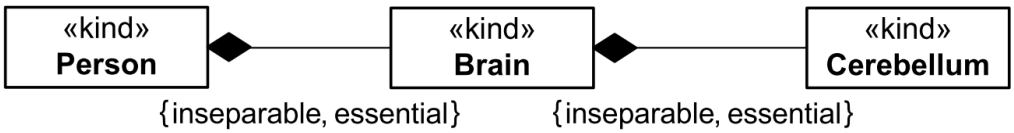


Figure 3.13 - Examples of (functional) essential parthood between complexes

4 OntoUML as a UML 2.0 Profile

In this chapter, we initially describe the UML 2.0 Profile mechanism which gives the ability to tailor the UML metamodel for different platforms or domains. Next, we perform an ontological interpretation of UML in a very similar way to (Guizzardi, 2005), i.e., we compare the concepts of the foundational ontology discussed in Chapter 3 with the semantics of standard UML constructs. During this process, we extend some of the concepts of UML to transform it into a well-founded conceptual modeling language and produce the OntoUML metamodel in terms of the UML Profile mechanism. This stage is done in a different way to (Guizzardi, 2005) because the specification provided by the author is not completely a lightweight extension. As a matter of fact, by the time in which his work was being finished, the UML 2.0 Superstructure specification was being released. Afterwards, we use our metamodel specification to implement our Front End graphical editor in the Eclipse tool. Finally, we address our modifications in the metamodel specification of (Guizzardi, 2005) in order to fit the UML lightweight extension mechanism.

4.1 UML Profiles

This section is based on the UML Superstructure documentation version 2.2 (OMG, 2009) and quotation marks related to this document will be omitted.

The *Profile* mechanism has been specifically defined for providing a *lightweight extension* mechanism to the UML standard. In previous versions of UML, stereotypes and tagged values were used as string-based extensions that could be attached to UML model elements in a flexible way. In subsequent revisions of UML, the notion of a Profile was defined in order to provide more structure and precision to the definition of *Stereotypes* and *Tagged values*. The UML 2.0 infrastructure and superstructure specifications have carried this further, by defining it as a specific meta-modeling technique. Stereotypes are specific metaclasses, tagged values are standard meta-attributes, and profiles are specific kinds of packages.

There are some requirements related to the profile semantics worth mentioning here. First, a profile must provide mechanisms for specializing a reference metamodel[3] in such a way that the specialized semantics do not contradict the semantics of the reference metamodel. That is, profile constraints may typically define well-formedness rules that are more constraining (but consistent with) those specified by the reference metamodel. In order to satisfy that, UML Profiles should form a metamodel extension mechanism that imposes certain restrictions on how the UML metamodel can be modified. The reference metamodel is considered as a "read only" model, that is extended without changes by profiles. It is therefore forbidden to insert new metaclasses in the UML metaclass

[3] In this particular section, we use the term *reference metamodel* in a different sense, i.e., unrelated to the one given in the context of our modeling infrastructure.

hierarchy (i.e., new super-classes for standard UML metaclasses) or to modify standard UML metaclass definitions (e.g., by adding meta-associations). Additionally, profiles can be dynamically applied or retracted from a model. It is possible on an existing model to apply new profiles, or to change the set of applied profiles.

The profiles mechanism is not a first-class extension mechanism (i.e., it does not allow for modifying existing metamodels). Rather, the intention of profiles is to give a straightforward mechanism for adapting an existing metamodel with constructs that are specific to a particular domain, platform, or method. Each such adaptation is grouped in a profile. It is not possible to take away any of the constraints that apply to a metamodel such as UML using a profile, but it is possible to add new constraints that are specific to the profile. The only other restrictions are those inherent in the profiles mechanism; there is nothing else that is intended to limit the way in which a metamodel is customized. In the sequel, we discuss some constructs related to the profile mechanism.

A profile is a kind of Package that extends a reference metamodel. The primary extension construct is the Stereotype, which is defined as part of Profiles. Once a profile has been applied to a package, it is allowed to remove the applied profile at will. Removing a profile implies that all elements that are instances of elements defined in a profile are deleted. The removal of an applied profile leaves the instances elements from the referenced metamodel intact. It is only the instances of the elements from the profile that are deleted. This means that, for example, a profiled UML model can always be interchanged with another tool that does not support the profile and be interpreted as a pure UML model.

A Stereotype defines how an existing metaclass may be extended, and enables the use of platform or domain specific terminology or notation. It is a kind of Class that extends Classes through Extensions. Each stereotype may extend one or more classes through extensions as part of a profile. Similarly, a class may be extended by one or more stereotypes. Just like a class, a stereotype may have properties, which may be referred to as tag definitions. When a stereotype is applied to a model element, the values of the properties may be referred to as *tagged values*. In UML 2.0, a tagged value can only be represented as an attribute defined on a stereotype. Therefore, a model element must be extended by a stereotype in order to be extended by tagged values.

An Extension is used to indicate that the properties of a metaclass are extended through a stereotype, and gives the ability to flexibly add (and later remove) stereotypes to classes. Extension is a kind of Association. One end of the Extension is an ordinary Property and the other end is an Extension End. The former ties the Extension to a Class, while the latter ties the Extension to a Stereotype that extends the Class. Extension has a derived boolean attribute called isRequired.

A required extension means that an instance of a stereotype must always be linked to an instance of the extended metaclass. The instance of the stereotype is typically deleted only when either the instance of the extended metaclass is deleted, or when the profile defining the stereotype is removed from the applied profiles of the package. The model is not well formed if an instance of the stereotype is not present when isRequired is true. If the extending stereotype has subclasses, then at most one instance of the stereotype or one of its subclasses is required.

A non-required extension means that an instance of a stereotype can be linked to an instance of an extended metaclass at will, and also later deleted at will; however, there is no requirement that each instance of a metaclass must be extended. An instance of a stereotype is further deleted when either the instance of the extended metaclass is deleted, or when the profile defining the stereotype is removed from the applied profiles of the package.

The notation for an Extension is an arrow pointing from a Stereotype to the extended Class, where the arrowhead is shown as a filled triangle. If isRequired is true, the property {required} is shown near the extension end.

Figure 4.1 shows an example of UML Profile. There, the same stereotype Clock can extend either the metaclass Component or the metaclass Class. It also shows how different stereotypes can extend the same metaclass. Furthermore, Figure 4.2 depicts an example of application of the profile of Figure 4.1 in a class diagram. In it, two stereotypes, Clock and Creator, are applied to the same model element. Note that the attribute values of each of the applied stereotypes are shown in a comment symbol attached to the model element.

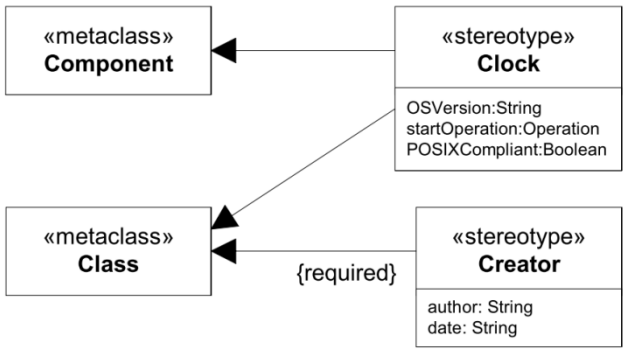


Figure 4.1 - Sample UML Profile (OMG, 2009)

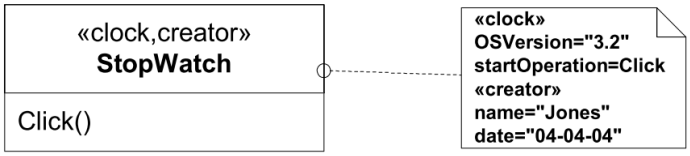


Figure 4.2 – Sample application of the Profile of Figure 4.1 (OMG, 2009)

4.2 Ontological Interpretation and Re-Design

In this section, we construct the OntoUML profile by investigating the semantics of the UML constructs and comparing it with the concepts of the foundation ontology of (Guizzardi, 2005), described in Chapter 3. If a certain concept has no corresponding construct, we have to create stereotypes in order to expand the power of expression of the language. This ontological interpretation was provided in (Guizzardi, 2005). However, the re-design of UML in a UML 2.0 Profile mechanism (i.e., involving Stereotypes and Metaclasses) is an original part of our work.

The first interpretation to be made is related to the UML metaclass *Class*. By Class, we mean the notion of first-order class (as opposed to powertypes) and one whose instances are single objects (as opposed to association classes, whose instances are tuples of objects). The ontological interpretation of a UML Class is that of a *Monadic Universal* (i.e., the supertype of Substantial Universals and Moment Universals). Since Substantials are prior to Moments from an identification point of view, it is typically the case that most classes in a class diagram should be interpreted as Substantial Universals. Because in UML there are no modeling constructs that represent the ontological categories specializing Substantial Universal, an extension of the UML metaclass Class is proposed in Figure 4.3 (there, Object Class is meant to be Substantial Universal). The extension relation between the UML metaclass Class and the stereotype Object Class allows every non-abstract stereotype in this hierarchy to be applied to Classes in a class diagram. Therefore, the stereotypes Kind, Quantity, Collective, SubKind, Phase, Role, Category, RoleMixin and Mixin are marked as non-abstract, while the others are abstract (thus, decorated in italics in the UML Profile specification).

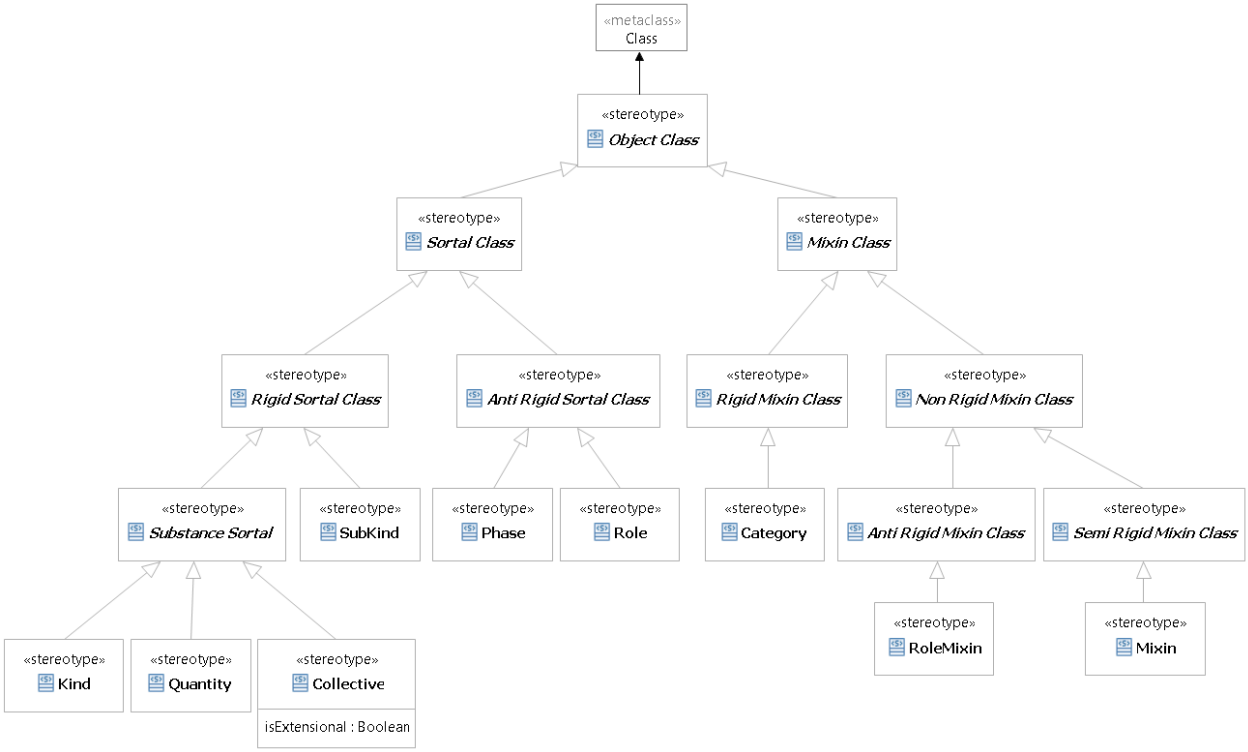


Figure 4.3 - Fragment of the UML Profile representing Substantial Universals

With respect to some Moment Universals, such as Mode Universals and Relator Universals, we have once more the same problem of ontological concepts prescribed in Guizzardi's theory that are not represented by any modeling construct in the language. Therefore, once again we extend the UML metaclass Class in order to represent those types of Moments (depicted in Figure 4.4).

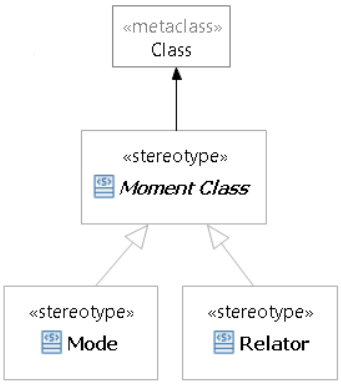


Figure 4.4 - Fragment of the UML Profile representing some Moment Universals

On the other hand, Quality Universals (which are also Moment Universals) are typically not represented in a conceptual model explicitly but via *attribute functions*. For example, suppose we have the substantial universal Apple whose instances exemplify the universal Weight. Thus, for an arbitrary instance *x* of Apple there is a quality *w* (instance of the quality universal Weight) that inheres in *x*. Associated with the universal Weight, there is a quality dimension *weightValue*, which is

a set isomorphic to the half line of positive integers obeying the same ordering structure. In this case, we can define an *attribute function* called *weight*(Kg), which maps each instance of apple (and in particular *x*) onto a point in a quality dimension, i.e., its *qualia*. Therefore, attribute functions are the ontological interpretation of UML attributes. Here, we prescribe that the associated types of attributes should be restricted to DataTypes only (as opposed to Classes), since we are considering that attributes should only be used to represent attribute functions.

The DataType associated with an attribute A of class C is the representation of the quality structure that is the co-domain of the attribute function represented by A. We can say that each field of a datatype should be interpreted as representing one of the integral dimensions of the quality domain represented by the datatype. Thus, a quality structure is the ontological interpretation of the UML DataType construct. Also, we propose to use navigable ends to represent only attribute functions whose co-domains are multidimensional quality structures (quality domains). Conversely, those functions whose co-domains are quality dimensions should only be represented by the attributes textual notation. The difference between the representation of attribute functions related to single and multidimensional quality structures is depicted in Figure 4.5.

Figure 4.5 - Representations of attribute functions (Guizzardi, 2005)

Finally, in order to represent Relations (material and formal), Dependency Relationships (mediation, characterization and derivation) and Meronymics (componentOf, subQuantityOf, subCollectionOf and memberOf), we need to extend the UML metaclass Association to overcome the construct incompleteness. There are some reasons to choose this metaclass as the point of extension. First, Associations carry a name and may be marked as derived, which are important aspects for Material and Formal Relations. Also, they relate types via Properties, so we are able to specify minimum and maximum cardinalities in each association end (which is important for all the types of relations), as much as specify an end as Read Only (which is important for Dependency Relationships). As a result, our extension is represented in Figure 4.6. Since Dependency Relationships and Part-Whole relations are always binary and directed, they can be related via an abstract stereotype called Directed Binary Association.

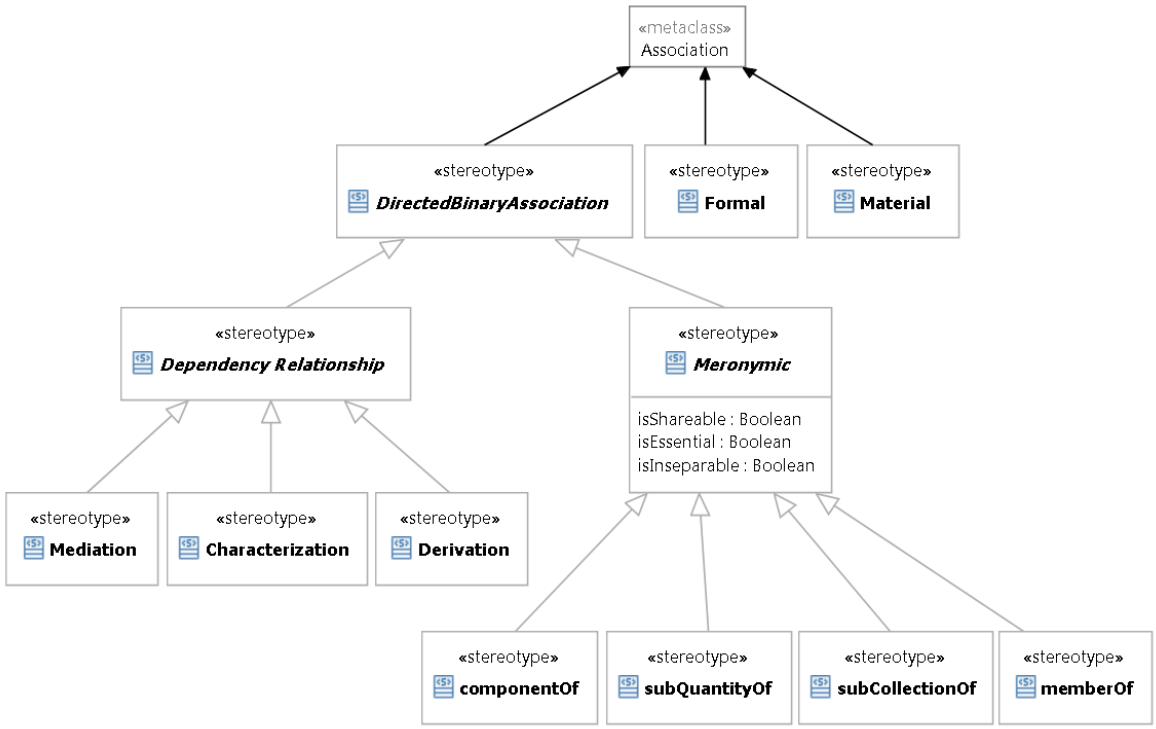


Figure 4.6 - Fragment of the UML Profile representing relations in general

4.3 The Front End Editor

When the fragments of Figure 4.3, Figure 4.4 and Figure 4.6 are combined, we obtain our OntoUML Profile specification. In the sequel, we use this specification in the Eclipse tool to obtain the front end graphical editor of our infrastructure. As a matter of fact, the standard UML graphical editor of Eclipse will allow the application of our Profile, once it has been defined. This situation is illustrated in Figure 4.7.

As soon as the profile is applied to a model, the graphical editor allows the user to decorate the standard UML constructs with stereotypes, based on the underlying metamodel implementation. In OntoUML, a user may decorate Classes and Associations. The behavior of our editor with respect to them is depicted in Figure 4.8 and Figure 4.9, respectively. Additionally, the user may remove a profile application and keep the standard UML model. This is an interesting aspect for interoperating/communicating with tools/people in terms of standard UML without further effort[4].

Moreover, one may notice that the editor exhibits a function for validation. Though it is possible to add constraints in the Profile specification and perform batch validation in the generated editor, due to the lack of documentation related to this feature and since our reference metamodel already supports batch validation, we postponed this aspect to future works.

[4] In theory, this process should reversible, i.e., the user could re-apply the profile and regain the applied stereotypes, but the current version of the Eclipse editor does not behave very well w.r.t. this feature.

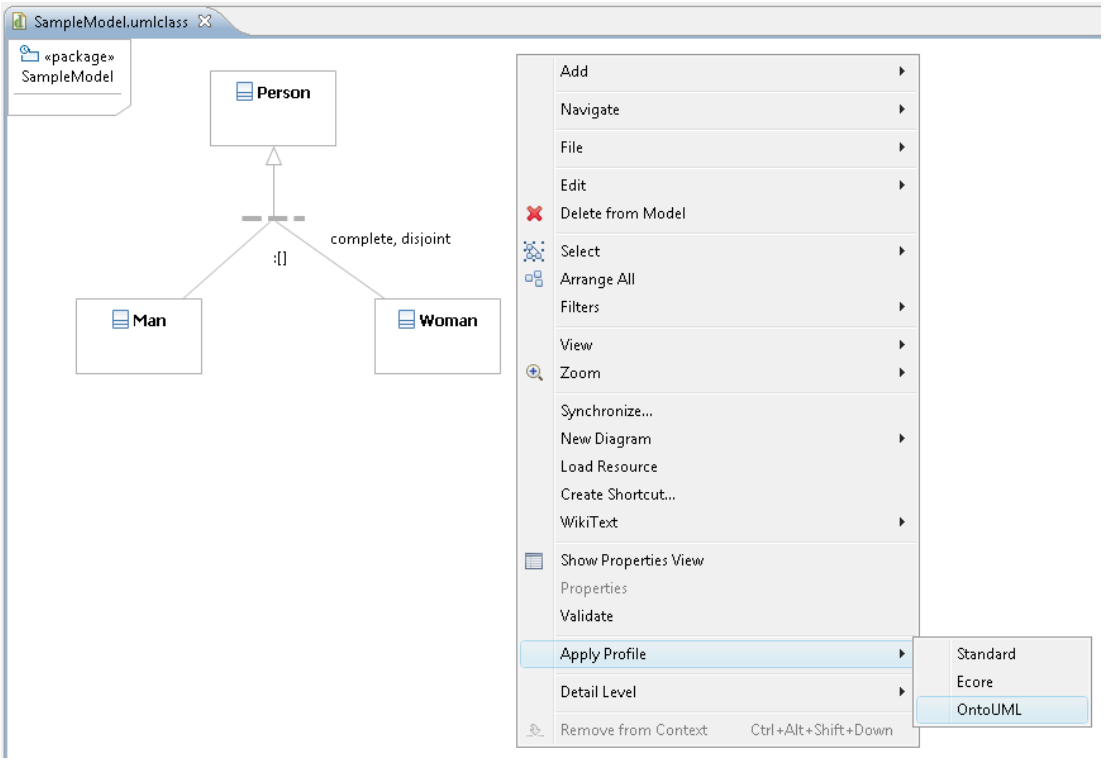


Figure 4.7 – Applying the OntoUML Profile to a standard UML model

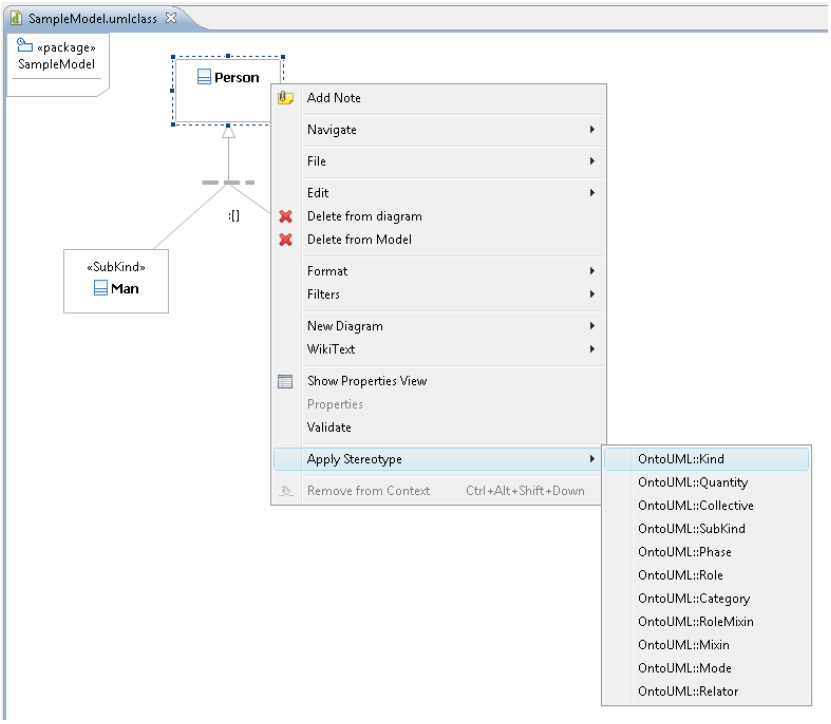


Figure 4.8 - Applying an OntoUML Stereotype to a Class

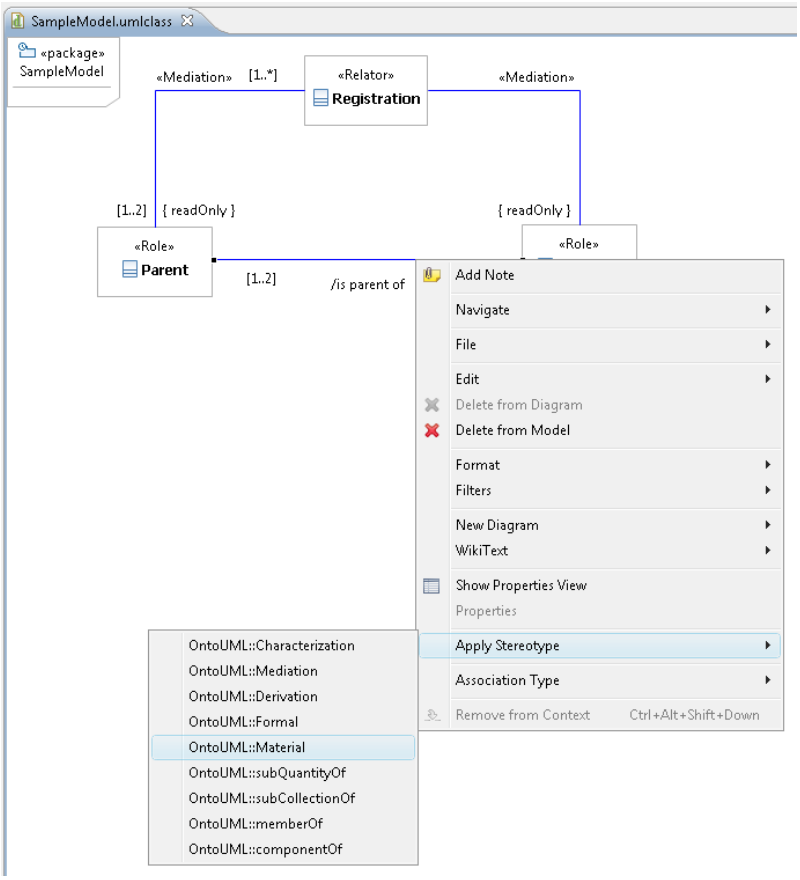


Figure 4.9 - Applying an OntoUML Stereotype to an Association

4.4 Differences from previous works

In (Guizzardi, 2005), the author provides various fragments of the revised UML metamodel while doing the ontological interpretation and re-design of the language. Nevertheless, the metamodel specification is not presented in terms of metaclasses and stereotypes, but just in terms of metaclasses with no further distinction between them. Thus, in Figure 4.10 we connected two of those fragments in order to explain the difference between our work and Guizzardi's work. Our discussion is based on the elements filled with grey background.

First, we need to make the distinctions among the elements in the fragment, defining which of them are UML metaclasses and which are stereotypes. Classifier is the UML metaclass that represents Classes, Associations and DataTypes. Association was already mentioned to be a UML metaclass that relates Classifiers via Properties, so they are able to specify cardinality constraints. In Guizzardi's thesis, only Material and Formal Relations were defined as stereotypes of the UML metaclass Association. Therefore, to represent Dependency Relationships and Meronymics, the author creates a stereotype called Directed Binary Relationship which is an extension of the UML metaclass Directed Relationship. In order to connect Material and Formal Relations with Dependency Relationships and Meronymics, he created another stereotype called Relational Classifier as a

supertype of Directed Binary Relationship and Association. Additionally, Relation Classifier is an extension of the UML metaclass Classifier.

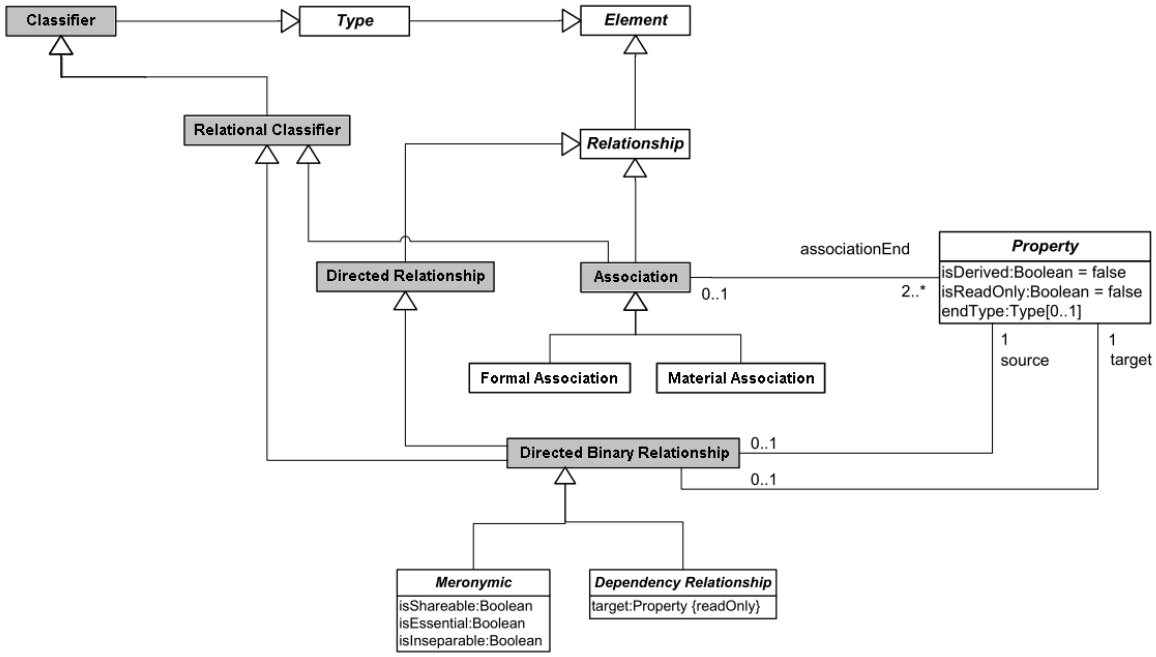


Figure 4.10 - Adapted fragment from (Guizzardi, 2005) involving relations in general

There are several problems with this approach. The first problem is related with the extension relation between the stereotype Directed Binary Relationship and the UML metaclass Directed Relationship. The Directed Relationship metaclass is an abstract metaclass, thus, we are not able to find a pure Directed Relationship in a class diagram to decorate with a stereotype. Then, analyzing the subtypes of this metaclass, we find the metaclasses Dependency, Element Import, Package Import, Generalization and Package Merge. As a consequence, one can only apply a stereotype related to Dependency Relationships (e.g., mediation) or Meronymics (e.g., componentOf) over one of those metaclasses previously mentioned. Notice that none of them is associated with cardinalities, since they simply relate elements directly (e.g., a generalization relates directly two Classifiers without the intermediate of Properties). It is true that, in the fragment of Figure 4.10, Directed Binary Relationship contains two meta-associations relating it to Properties (source and target), but this procedure (namely, addition of meta-associations) is invalid in a lightweight extension.

Furthermore, the problem of not having appropriate concrete elements in a class diagram to apply the stereotypes related to Dependency Relationship and Meronymics may apparently be solved by the stereotype Relational Classifier. This happens because a Directed Binary Relationship is a Relation Classifier which is, in its turn, a Classifier. Thus, this allows the stereotype Directed Binary Relationship to be applied to any Classifier in a class diagram. That would allow Directed Binary Relationships to be applied to Associations. But, certainly, it is inadequate to allow the user to apply,

for example, the stereotype memberOf to a Class or a DataType. Anyhow, the relation between the UML metaclass Association and the stereotype Relational Classifier is illegal, since a stereotype extends a metaclass, not the opposite. For this reason, we provided a different approach that causes all the relations in general to be extensions of the UML metaclass Association.

5 THE REFERENCE METAMODEL

In this chapter, we construct the back end reference metamodel of our infrastructure. First, we discuss some aspects of the technology used to implement it, then we explain the metamodel itself. In the end, we expose the differences between our metamodel and the former metamodel defined in a previous work.

5.1 THE ECLIPSE MODELING FRAMEWORK

The Eclipse Modeling Framework (EMF) (The Eclipse Foundation, 2010) (Steinberg, et al., 2009) is a modeling framework that exploits the facilities provided by Eclipse. EMF relates modeling concepts directly to their implementations, thereby bringing to Eclipse – and Java Developers in general – the benefits of modeling with a low cost of entry. More deeply, EMF is a code generation facility that allows the user to define a model in a specific form (such as annotated Java interfaces, XML or UML) and then generate any of the others and also the corresponding implementation classes. An EMF model is the common high-level representation that intermediates all this different forms of representation. Therefore, EMF is a framework for describing a model and then generating other artifacts from it.

Models in EMF are represented in a *metamodeling* language called Ecore, i.e., Ecore is a language for defining *metamodels* (like the OntoUML metamodel, for example) that, on the other hand, will serve to create *models* (such as an OntoUML model). A simplified subset of the Ecore metamodel is shown in Figure 5.1. *EClass* is used to represent a modeled class; it has a name and zero or more *EStructuralFeatures*, which can be *EReferences* or *EAttributes*. An *EAttribute* is used to represent a modeled attribute of the *EClass*; it has a name and a type. An *EReference* is used to represent an association between the *EClass* under discussion and another *EClass*; it has a name, a reference type that represents the other *EClass,* the cardinalities of the association, and a boolean flag to indicate if it represents containment. Finally, *EDataType* is used to represent the type of an attribute; it can be a primitive type like *int* or *float* or an object type like *java.util.Date*.

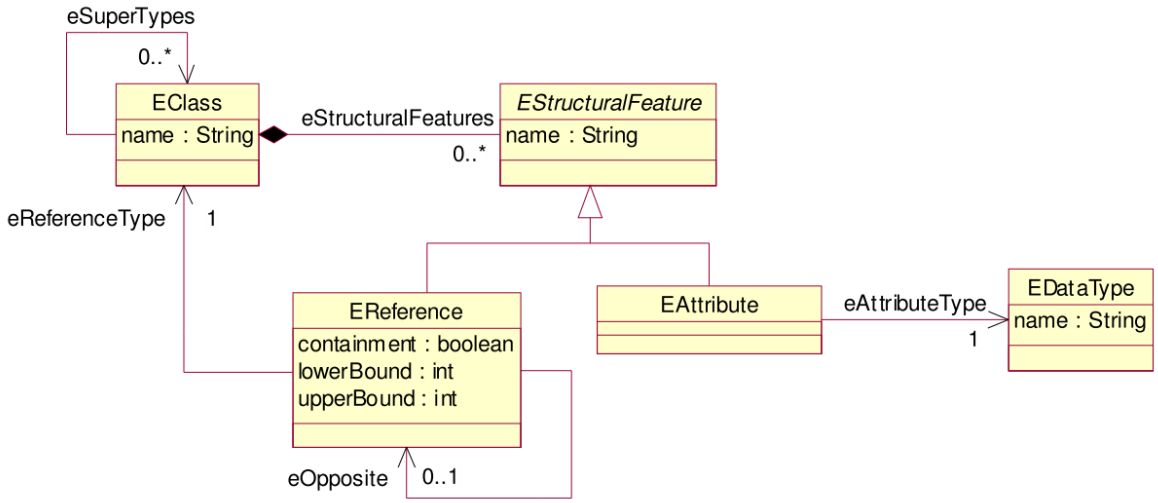


Figure 5.1 - A simplified subset of the Ecore metamodel (Steinberg, 2008)

Take, for example, the UML model on top of Figure 5.2 representing purchase orders for some store or supplier. One can use the instances of the classes defined in the Ecore metamodel to describe this model. The purchase order class is represented as an instance of *EClass* named "PurchaseOrder". It contains two attributes (instances of *EAttribute* that are accessed via *eStructuralFeatures*) named "shipTo" and "billTo", and one reference (an instance of *EReference* that is also accessed via *eStructuralFeatures*) named "items", for which *eReferenceType* is equal to another *EClass* instance named "Item". These instances are shown on the bottom of Figure 5.2. (Steinberg, et al., 2009)

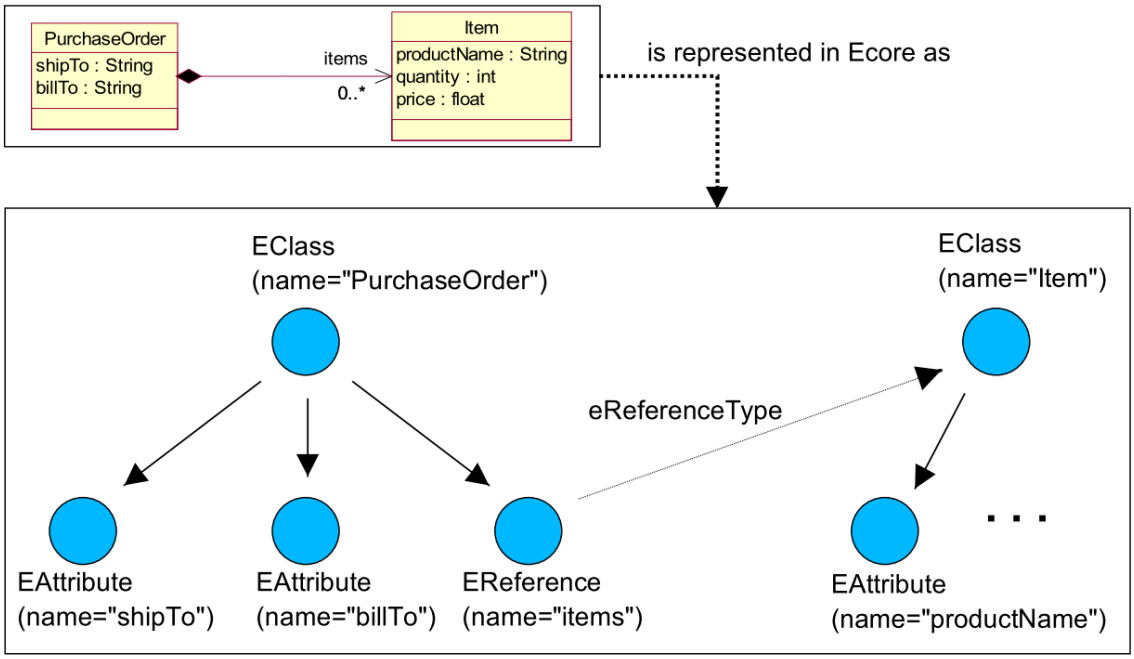


Figure 5.2 - An Ecore Model as an instance of the Ecore Metamodel (Boldt, et al., 2006)

Besides obtaining an Ecore Model from a UML model, an XML schema or annotated Java interfaces, it is possible to edit Ecore models using a simple tree-based editor included in EMF or a graphical tool based on UML notation included in Eclipse Modeling. This is the method used in this work.

An important benefit of EMF is the boost in productivity that results from automatic code generation. With the Ecore model of Figure 5.2, it is possible to generate the Java classes for Purchase Order and Item, including the attributes and references specified, according to some patterns. Also, based on the Ecore model, it is possible to generate viewers and editors that will display and edit (i.e., copy, paste, drag-and-drop, etc) instances of the Ecore Model. Figure 5.3 depicts the editor generated for the Purchase Order (meta)model. There, we are able to see a single Purchase Order called “Xmas’ Gifts” that contains two Items, named “EMF Book” and “Wii”. The attributes of the selected Item, viz. “Wii”, are shown in the properties tab (only Product Name, Quantity and Price were exhibited in the (meta)model of Figure 5.2). Therefore, with EMF we are able to build our *metamodel* using Ecore and then generate an editor for creating our *models* (based on our metamodel).

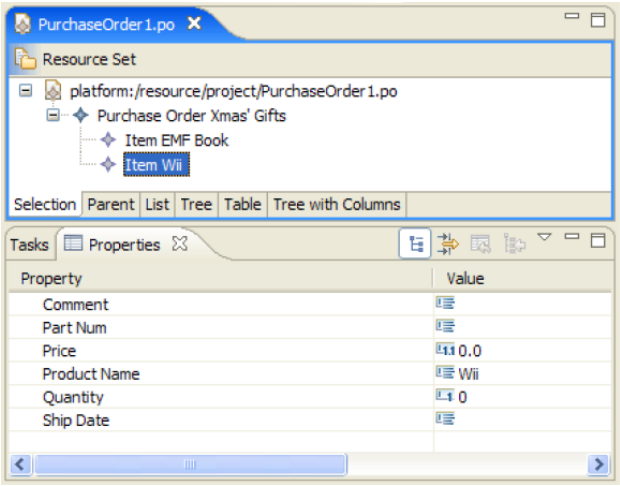


Figure 5.3 - Editor representing an instance of the Purchase Order (meta)model (Boldt, et al., 2006)

Additionally, EMF has a *validation framework* that provides means to declare constraints in an Ecore model to be verified when a validation is performed against the instances of the model. This validation can be invoked both from Java code or on the generated editors. These constraints are defined via annotations on the elements of the Ecore model and affect the generated code. Initially, the validation framework is only capable of generating some structure that helps the implementation of the constraints and the Java code still has to be hand modified to make them work. Nevertheless, by integrating EMF / Java Emitter Templates (JET) with the Modeling Development Tools (MDT), it is possible to directly obtain the Java code for validation, without any post-generation custom code (Damus, 2007). With this approach, we make use of the MDT OCL parser/interpreter technology to

embed OCL constraints in an Ecore model via annotations. Then, the generated model classes will have methods implementing these constraints. They can be invariant constraints, derived attributes/references and operations.

5.2 OntoUML in Ecore

In order to implement the OntoUML metamodel in Ecore, we take the Eclipse implementation of the UML Superstructure (included in EMF 2.5) as a foundation. Since the UML Superstructure includes metaclasses from many UML diagrams (such as Class, Use Case, State Machine and Sequence Diagrams), our first activity is to identify which metaclasses are convenient for OntoUML. Since OntoUML is a profile related to the UML class diagram, most of the metaclasses related to it are preserved. Exceptions are AssociationClass (since Guizzardi claims it is an ambiguous construct); BehavioralFeature, BehavioredClassifier, Operation, Parameter, ParameterDirectionKind (since the foundational ontology is related to structural aspects only); Usage, Abstraction, Realization and Substitution (subtypes of Dependency related to implementation details); Interface and InterfaceRealization (design and implementation constructs). Additionally, all the metaclasses that belong exclusively to diagrams different from the class diagram are removed from our Ecore model.

In the sequence, with a custom UML class diagram, we extend the UML metaclasses in the exact same way we did when implementing the UML Profile mechanism. The only difference is that there is no distinction between UML metaclasses and Profile stereotypes and also between generalizations and extensions. Since we are dealing with an Ecore model, all the elements of the metamodel are instances of *EClass* forming hierarchies via *eSuperTypes*. Hence, we abstain from showing the analogous fragments of the metamodel related to OntoUML.

Once the metaclasses related to OntoUML are added to the metamodel, the final step is to add annotations in the metaclasses in order to express constraints that restrict the way OntoUML models are built. Figure 5.4 depicts a fragment of the OntoUML metamodel in the Ecore tree editor. The pattern for OCL invariant constraints requires one *EAnnotation* named “Ecore” containing a list of constraint labels (e.g., RoleConstraint2) and another *EAnnotation* named “OCL” containing the implementation of each constraint. The *EAnnotation* “Comments” has no effect in the Java code generation but we use it for the sake of clarity, since a constraint label such as RoleConstraint2 is not very informative. Full details of the OCL constraints are provided in Chapter 6.

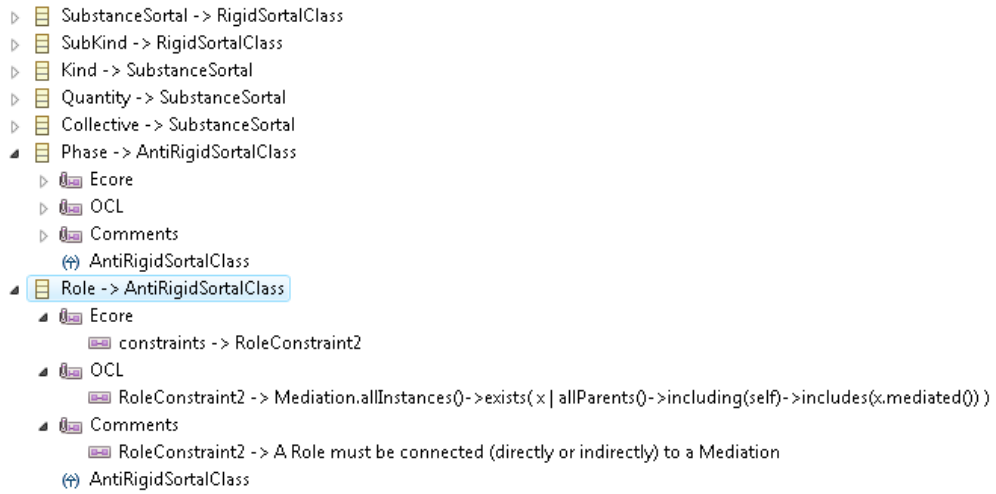


Figure 5.4 – Fragment of the OntoUML metamodel with constraints in the Ecore tree editor

As mentioned before, from an Ecore model it is possible to generate not just Java code but also a customizable tree editor. Thus, by generating the default tree editor, we obtain a way to visualize (and also edit) our back end metamodel. This is depicted in Figure 5.5. Of course, it is not the main purpose to view and edit the back end metamodel, since it is the front end tool's responsibility. Nonetheless, the greatest advantage of this generated editor is to provide a way to validate the model elements graphically. The user may select any model element, right-click on it and request a validation, then the framework will check if any of the constraints is violated in the selected element or in any of its parts. If any constraint is violated, the user will be informed of the element in question and the constraint label (see Figure 5.6). To validate the whole model is just a matter of validating the ultimate container (such as the Package "OntoUML Sample" in Figure 5.5).

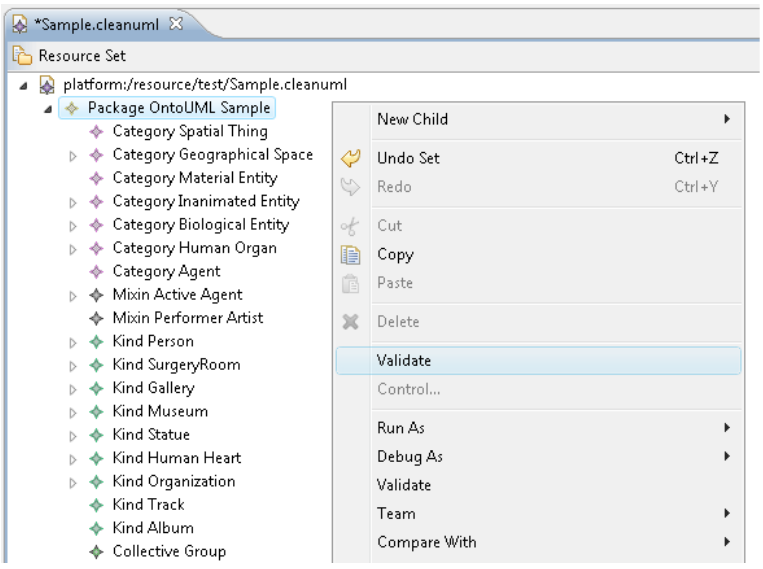


Figure 5.5 - The tree editor generated from the back end metamodel

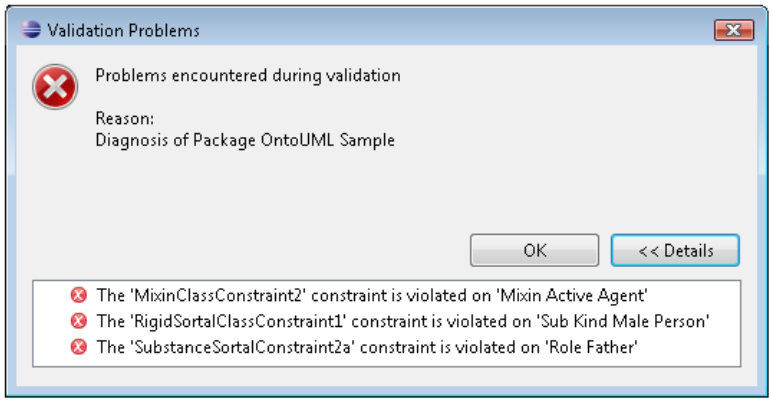


Figure 5.6 - Validation on the OntoUML tree editor

5.3 DIFFERENCES FROM PREVIOUS WORKS

In this section, we address the differences between our reference metamodel and the metamodel of the GMF Editor defined in (Benevides, 2010), both implemented in Ecore.

The first difference is based on the discussion about whether Meronymic and Dependency Relationship should be extensions of UML Association or UML Directed Relationship (mentioned in Chapter 4, section 4.4). This happens because Benevides' metamodel is based on the specification provided in (Guizzardi, 2005). Therefore, our metamodel has Association as the supertype of Meronymic and Dependency Relationship, while Benevides uses Directed Relationship as the supertype. Also, we abolish the OntoUML metaclass RelationalClassifier (also discussed in section 4.4).

Another difference lies on the fact that Benevides' approach simplifies the UML metamodel. As a consequence, his metamodel does not possess the concept of Package and all the related concepts like Packageable Element, Package Import, Package Merge, Namespace, Element Import and Model. His metamodel uses a metaclass called "Container" to encompass all the Elements of the model (such as Classes, Associations, DataTypes, GeneralizationSets, Generalizations, Properties and basically anything in a class diagram). On the other hand, in UML, the role of this "Container" is actually given by a Package, which contains Packageable Elements. Not every Element is a Packageable Element, thus, not every instance of the metaclass Element (the ultimate ancestor of all the other UML metaclasses) can be contained directly in a Package. Relevant examples of Packageable Elements include: Package, Type (the supertype of Classifier which, in its turn, is the supertype of Class, Association and DataType), GeneralizationSet and Dependency. This means that Elements like Generalization and Property (able to be contained in Benevides' "Container") cannot be directly contained in a Package. A Generalization is owned by the specific Classifier and a Property is owned by the Classifier that has it as an attribute. Accordingly, by using the concept of Package and Packageable Elements, we not only allow our OntoUML models to be modularized but also we apply

the rules of containment between the UML metaclasses in a straightforward way. Figure 5.7 illustrates some of the aspects discussed about Packages.

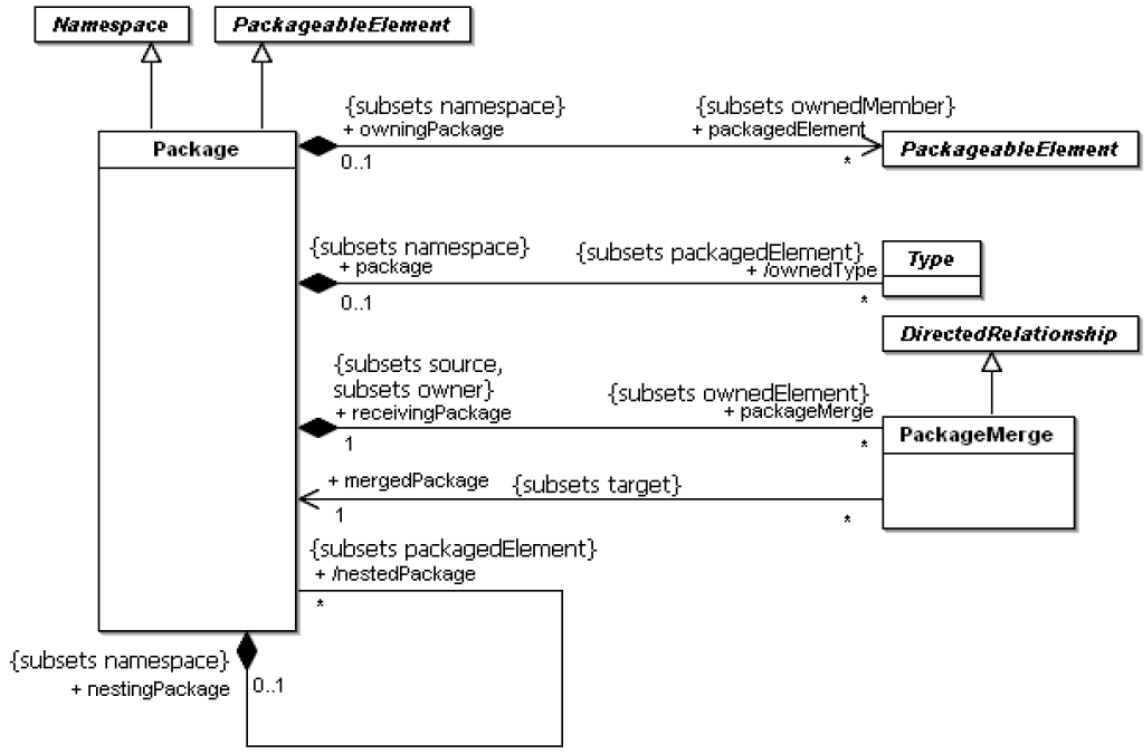


Figure 5.7 - Fragment of the UML metamodel related to Packages (OMG, 2009)

Moreover, there are more concepts absent in his metamodel, namely, Constraint, Instance Specification, Slot, ValueSpecification (and its subtypes, InstanceValue, OpaqueExpression, LiteralSpecification, LiteralBoolean, LiteralString, LiteralInteger, LiteralUnlimitedNatural, LiteralNull, Expression, StringExpression), Dependency, Comment, Primitive Type, Enumeration and EnumerationLiteral. To illustrate another simplification of the UML metamodel, the metaclass ValueSpecification is important for specifying the minimum and maximum cardinalities of a MultiplicityElement (one of the supertypes of Property). The fragment of the metamodel concerning this situation is depicted in Figure 5.8. Since a Property is always related to a LiteralInteger and a LiteralUnlimitedNatural (particular types of ValueSpecification), Benevides specifies that a MultiplicityElement has two integer attributes (lower and upper) instead of relating it to two ValueSpecifications.

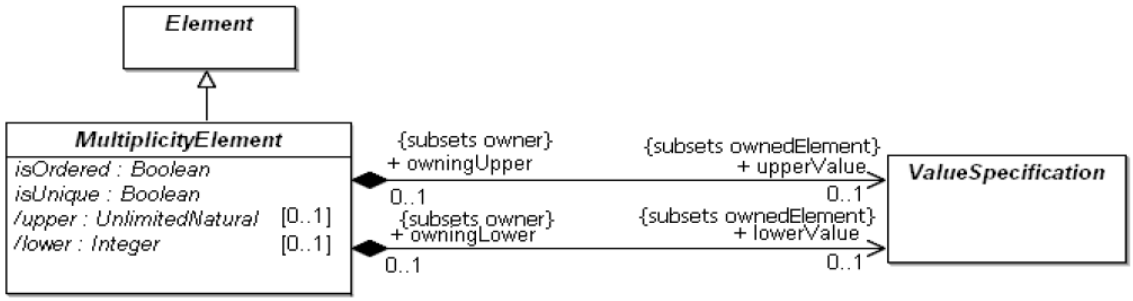


Figure 5.8 - Minimum and maximum cardinalities of a Property in UML (OMG, 2009)

Like mentioned before, Benevides was forced to incorporate concrete syntax elements in the GMF Editor's metamodel in order to fully implement the GMF Editor. For instance, Figure 5.9 depicts a fragment of the GMF Editor's metamodel related to Association and DirectedRelationship. In this fragment, all the EAttributes and EReferences containing "aux" in their names are auxiliary for achieving concrete syntax[5]. Our reference metamodel, in contrast, does not contain constructs of this type, since it is independent of editing tools.

Figure 5.9 – Auxiliary constructs in the GMF Editor's metamodel to achieve concrete syntax

Finally, the last difference is in respect to the place in which the OCL constraints are located. As mentioned on the previous section, in our work, all the constraints are embedded in the Ecore reference metamodel. In contrast, the Ecore metamodel used by the GMF Editor includes only some

[5] Except for "associationEnd3Aux" which is, perhaps, equivalent to the "ownedEnd" meta-association defined in the UML documentation. That meta-association specifies that an Association (possibly) owns the Properties that it is related to.

derived attributes and auxiliary operations. The constraints used in live and batch validation are incorporated in a file related to the GMF infrastructure (more specifically, a GMF map). Thus, not only the abstract syntax representation is harmed by GMF, but also the validation framework. This happens because validation can only take place in the context of the GMF graphical editor, as opposed to our approach, in which it happens out of the context of a front end tool (at least in the case of batch validation, since live validation should happen during model editing).

6 OntoUML Syntactical Constraints

This chapter contains the set of constraints, derived from the postulates of the foundational ontology, that restrict the ways the elements can be related. The goal is to have a metamodel such that all syntactically correct specifications according to it have logical models that are *intended world structures* of the conceptualizations they are supposed to represent. Those constraints are written in the Object Constraint Language (OCL) (OMG, 2006).

Most of the OCL implementation is different from the previous work of (Benevides, 2010) due to optimizations and modularizations that were made, differences between the OntoUML metamodels and also due to some conceptual changes. Nonetheless, we content ourselves to present the constraints as a reference summary first and provide justifications later. Therefore, constraints that will be target of further discussion are marked with a star symbol (*).

6.1 Generalizations

This section is dedicated to the subtype relation between Universals. It makes use of two standard UML operations of Classifiers, namely *parents* and *allParents*. The former returns only the direct parents of the Classifier while the latter recursively returns all the parents in the model.

Ultimate Substance Sortal

Every Object Class must not have more than one Substance Sortal ancestor

```
context ObjectClass inv:
allParents()->select( x | x.oclIsKindOf(SubstanceSortal) )->size() <= 1
```

Concrete Sortal has Substance Sortal ancestor (*)

Every non-abstract Sortal must have a Substance Sortal ancestor (or be a Substance Sortal)

```
context SortalClass inv:
not isAbstract and not oclIsKindOf(SubstanceSortal)
implies
allParents()->exists( x | x.oclIsKindOf(SubstanceSortal) )
```

Substance Sortal does not specialize Rigid Sortal

A Substance Sortal cannot have a Rigid Sortal parent

```
context SubstanceSortal inv:
parents()->select( x | x.oclIsKindOf(RigidSortalClass) )->isEmpty()
```

Rigid Sortal does not specialize Anti-Rigid

A Rigid Sortal cannot have an Anti-Rigid parent (Role, Phase and RoleMixin)

```
context RigidSortalClass inv:
self.parents()->select( x | x.oclIsKindOf(AntiRigidSortalClass) or x.oclIsKindOf(RoleMixin) )->isEmpty()
```

Mixin Class does not specialize Sortal Class

A Mixin Class (Category, Mixin, RoleMixin) cannot have a Sortal parent (Kind, Quantity, Collective, SubKind, Phase, Role)

```
context MixinClass inv:
parents()->select( x | x.oclIsKindOf(SortalClass) )->isEmpty()
```

Category does not specialize RoleMixin (*)

A Category cannot have a Role Mixin parent

```
context Category inv:
parents()->select( x | x.oclIsTypeOf(RoleMixin) )->isEmpty()
```

Mixin does not specialize RoleMixin

A Mixin cannot have a RoleMixin parent

```
context Mixin inv:
parents()->select( x | x.oclIsTypeOf(RoleMixin) )->isEmpty()
```

Object Class specializes Object Class (*)

An Object Class only participates in a Generalization with another Object Class

```
context Generalization inv:
(specific.oclIsKindOf(ObjectClass) implies general.oclIsKindOf(ObjectClass))
and
(general.oclIsKindOf(ObjectClass) implies specific.oclIsKindOf(ObjectClass))
```

Table 6.1 contains a summary of generalizations involving Object Classes. For example, as a consequence of two constraints, Substance Sortals can only specialize Categories and Mixins. In contrast, there are no constraints that forbid Anti-Rigid Sortals from specializing other Object Classes.

	Object Class								
	Sortal Class						Mixin Class		
	Rigid				Anti-Rigid		Rigid	Non-Rigid	
	Substance Sortal			SubKind	Role	Phase	Category	Semi-Rigid	Anti-Rigid
	Kind	Quantity	Collective					Mixin	RoleMixin
Kind	-	-	-	-	--	--	X	X	--
Quantity	-	-	-	-	--	--	X	X	--
Collective	-	-	-	-	--	--	X	X	--
SubKind	X	X	X	X	--	--	X	X	--
Role	X	X	X	X	X	X	X	X	X
Phase	X	X	X	X	X	X	X	X	X
Category	---	---	---	---	---	---	X	X	=
Mixin	---	---	---	---	---	---	X	X	= =
RoleMixin	---	---	---	---	---	---	X	X	X

Table 6.1 - Generalization Constraints

Symbol	Description
-	Substance Sortal does not specialize Rigid Sortal
--	Rigid Sortal does not specialize Anti-Rigid
---	Mixin Class does not specialize Sortal Class
=	Category does not specialize RoleMixin
= =	Mixin does not specialize RoleMixin
X	(Line) is able to specialize (Column)

6.2 Directed Binary Associations

This section is dedicated to Directed Binary Associations, viz. Dependency Relationships and Meronymics.

Directed Binary Association is binary (*)

Directed Binary Associations are always binary.

```
context DirectedBinaryAssociation inv: memberEnd->size() = 2
```

Before we proceed, we define some operations to help the implementation of the Dependency Relationship and Meronymic constraints.

Source End & Target End (*)

Auxiliary for Directed Binary Associations

context DirectedBinaryAssociation :: sourceEnd() : Property
body: memberEnd->at(1)

context DirectedBinaryAssociation :: targetEnd() : Property
body: memberEnd->at(2)

Whole End, Part End, Whole & Part (*)

Auxiliary for the Meronymic Relation

context Meronymic :: wholeEnd() : Property
body: sourceEnd()

context Meronymic :: partEnd() : Property
body: targetEnd()

context Meronymic :: whole() : Classifier
body: wholeEnd().type

context Meronymic :: part() : Classifier
body: partEnd().type

Relator End, Mediated End, Relator & Mediated (*)

Auxiliary for the Mediation Relation

context Mediation :: relatorEnd() : Property
body: sourceEnd()

context Mediation :: mediatedEnd() : Property
body: targetEnd()

context Mediation :: relator() : Classifier
body: relatorEnd().type

context Mediation :: mediated() : Classifier
body: mediatedEnd().type

Mode End, Characterized End, Mode & Characterized (*)

Auxiliary for the Characterization Relation

context Characterization :: modeEnd() : Property
body: sourceEnd()

context Characterization :: characterizedEnd() : Property
body: targetEnd()

context Characterization :: mode() : Classifier
body: modeEnd().type

context Characterization :: characterized() : Classifier
body: characterizedEnd().type

Material End, Relator End, Material & Relator (*)

Auxiliary for the Derivation Relation

context Derivation :: materialEnd() : Property
body: sourceEnd()

context Derivation :: relatorEnd() : Property
body: targetEnd()

context Derivation :: material() : Classifier
body: materialEnd().type

context Derivation :: relator() : Classifier
body: relatorEnd().type

6.2.1 DEPENDENCY RELATIONSHIP

The following constraints are related to Dependency Relationships (Mediation, Characterization, Derivation).

Dependency Relationship's source (*)

The source end minimum cardinality must be greater of equal to 1.

```
context DependencyRelationship inv: sourceEnd().lower >= 1
```

Dependency Relationship's target (*)

The target end is read only.

```
context DependencyRelationship inv: targetEnd().isReadOnly
```

Characterization's source

(Characterization) The source must be a Mode

```
context Characterization inv: mode().oclIsTypeOf(Mode)
```

Mediation's source

(Mediation) The source must be a Relator

```
context Mediation inv: relator().oclIsTypeOf(Relator)
```

Derivation's source

(Derivation) The source must be a Material Association

```
context Derivation inv: material().oclIsTypeOf(MaterialAssociation)
```

Derivation's target

(Derivation) The target must be a Relator

```
context Derivation inv: relator().oclIsTypeOf(Relator)
```

Mediation's target cardinality

The Mediated end minimum cardinality must be greater or equal to 1

```
context Mediation inv: mediatedEnd().lower >= 1
```

Characterization's target cardinality

The Characterized end cardinality is exactly one.

context Characterization **inv**: characterizedEnd().lower = 1 **and** characterizedEnd().upper = 1

Derivation's target cardinality

The Relator end cardinality is exactly one

context Derivation **inv**: relatorEnd().lower = 1 **and** relatorEnd().upper = 1

Relator & Mediation

A Relator must be connected (directly or indirectly) to a Mediation

context Relator **inv**:
Mediation.allInstances()->exists(x | allParents()->including(self)->includes(x.relator()))

Relator & Mediation 2

The sum of the minimum cardinalities of the mediated ends must be greater or equal to 2

context Relator **inv**:
Mediation.allInstances()->select(x | allParents()->including(self)->includes(x.relator()))->
collect (y | y.mediatedEnd().lower)->sum() >= 2

Role & Mediation (*)

A Role must be connected (directly or indirectly) to a Mediation

context Role **inv**:
Mediation.allInstances()->exists(x | allParents()->including(self)->includes(x.mediated()))

RoleMixin & Mediation (*)

A RoleMixin must be connected (directly or indirectly) to a Mediation

context RoleMixin **inv**: Mediation.allInstances()->exists(x | allParents()->including(self)->
includes(x.mediated()))

Mode & Characterization

A Mode must be connected (directly or indirectly) to a Characterization

context Mode **inv**:
Characterization.allInstances()->exists(x | allParents()->including(self)->includes(x.mode()))

Material Association & Derivation

Every Material Association must be connected to exactly one Derivation

```
context MaterialAssociation inv: Derivation.allInstances()->one( x | x.material() = self )
```

6.2.2 MERONYMIC

In this section we describe the constraints involving part-whole relations. In order to speak about the "nature" of the relata of part-whole relations, we need to define some auxiliary methods first.

Nature of Sortal

The three methods below check the type of the ultimate substance sortal that supplies the principle of identity to a specific sortal. If the sortal is a Kind or has a Kind ancestor, then informally we say that it has a Kind "nature".

```
context Classifier :: hasKindAncestor() : EBoolean
body: allParents()->including(self)->exists ( x | x.oclIsKindOf (Kind) )

context Classifier :: hasQuantityAncestor() : EBoolean
body: allParents()->including(self)->exists ( x | x.oclIsKindOf (Quantity) )

context Classifier :: hasCollectiveAncestor() : EBoolean
body: allParents()->including(self)->exists ( x | x.oclIsKindOf (Collective) )
```

Nature of Mixin Class (*)

The three methods below check if the complete offspring of a Mixin Class has a pure Kind, Quantity or Collective "nature".

```
context Classifier :: hasKindOffspring () : EBoolean
body: ObjectClass.allInstances()->select ( x | x.allParents()->includes(self) )->
forAll ( y | not y.hasQuantityAncestor() and not y.hasCollectiveAncestor() )

context Classifier :: hasQuantityOffspring() : EBoolean
body: ObjectClass.allInstances()->select ( x | x.allParents()->includes(self) )->
forAll ( y | not y.hasKindAncestor() and not y.hasCollectiveAncestor() )

context Classifier :: hasCollectiveOffspring() : EBoolean
body: ObjectClass.allInstances()->select ( x | x.allParents()->includes(self) )->
forAll ( y | not y.hasKindAncestor() and not y.hasQuantityAncestor() )
```

Nature of Relata

Finally, as a conjunction of the two categories of operations defined above, we are able to identify the "nature" of any Object Class.

```
context Classifier :: hasFunctionalComplexInstances() : EBoolean
body:
if oclIsKindOf (SortalClass) then
        hasKindAncestor()
else
        if oclIsKindOf (MixinClass) then
                hasKindOffspring()
        else
                false
        endif
endif

context Classifier :: hasQuantityInstances() : EBoolean
body:
if oclIsKindOf (SortalClass) then
        hasQuantityAncestor()
else
        if oclIsKindOf (MixinClass) then
                hasQuantityOffspring()
        else
                false
        endif
endif

context Classifier :: hasCollectiveInstances() : EBoolean
body:
if oclIsKindOf (SortalClass) then
        hasCollectiveAncestor()
else
        if oclIsKindOf (MixinClass) then
                hasCollectiveOffspring()
        else
                false
        endif
endif
```

Weak Supplementation

The sum of the minimum cardinalities of the part ends must be greater or equal to 2

context Meronymic **inv**:
Meronymic.allInstances()->select(x | x.whole() = whole())->collect(y | y.partEnd().lower)->sum() >= 2

Extensional Collective

All the parts of an extensional Collective are essential

context Collective **inv**:
isExtensional **implies** Meronymic.allInstances()->forAll(x | x.whole() = self implies x.isEssential)

componentOf relata

ComponentOf relates individuals that are functional complexes

context componentOf **inv**: whole().hasFunctionalComplexInstances()
context componentOf **inv**: part().hasFunctionalComplexInstances()

subQuantityOf relata

SubQuantityOf relates individuals that are quantities

context subQuantityOf **inv**: whole().hasQuantityInstances()
context subQuantityOf **inv**: part().hasQuantityInstances()

subCollectionOf relata

SubCollectionOf relates individuals that are collectives

context subCollectionOf **inv**: whole().hasCollectiveInstances()
context subCollectionOf **inv**: part().hasCollectiveInstances()

memberOf relata

MemberOf relates individuals that are functional complexes or collectives as parts of individuals that are collectives

context memberOf **inv**: whole().hasCollectiveInstances()
context memberOf **inv**: part().hasCollectiveInstances() **or** part().hasFunctionalComplexInstances()

subCollectionOf cardinality

The maximum cardinality of the part end is equal to 1

```
context subCollectionOf inv: partEnd().upper = 1
```

subQuantityOf cardinality

The maximum cardinality of the part end is equal to 1

```
context subQuantityOf inv: partEnd().upper = 1
```

subQuantityOf meta-properties

A part is always non-shareable

```
context subQuantityOf inv: not isShareable
```

A part is always essential

```
context subQuantityOf inv: isEssential
```

memberOf with essential parthood (*)

MemberOf with essential parthood implies an extensional whole

```
context memberOf inv:
isEssential
implies
if whole().oclIsKindOf (MixinClass) then
        ObjectClass.allInstances()->select( x | x.allParents()->includes(self) )->
        forAll( y | y.oclIsKindOf(Collective) implies y.oclAsType(Collective).isExtensional )
else
        whole().allParents()->including(whole())->
        forAll( x | x.oclIsKindOf(Collective) implies x.oclAsType(Collective).isExtensional )
endif
```

6.3 MISC

Finally, in this section, we include constraints that do not fit in any of the categories above.

Material Association's minimum cardinality

The minimum cardinality of every end must be greater or equal to 1

```
context MaterialAssociation inv: memberEnd->forAll( x | x.lower >= 1 )
```

Material Association is derived

```
context MaterialAssociation inv: isDerived
```

Mixin Class is abstract

```
context MixinClass inv: isAbstract
```

Phase Partition

A Phase of a Substance Sortal must be grouped in exactly one {disjoint, complete} Generalization Set with other Phases

```
context Phase inv:
let
	gsets : Bag(GeneralizationSet) =
	generalization->select( x | x.general.oclIsKindOf(SubstanceSortal) ).generalizationSet
in
	if gsets->size() = 1 then
		let
			gs : GeneralizationSet = gsets->any(true)
		in
			gs.generalization.specific->forAll( x | x.oclIsKindOf(Phase) )
			and
			gs.generalization->size() > 1 and gs.isDisjoint and gs.isCovering
	else
		false
	endif
```

6.4 DIFFERENCES FROM PREVIOUS WORKS

For now, we describe the differences between the constraints defined in this work and the previous works. Most of the constraints have implementation differences due to optimizations made and due to differences between metamodels. However, we focus our discussion in constraints with conceptual differences or related to the usability of the validation tool.

The 'Concrete Sortal has Substance Sortal Ancestor' was previously defined as a constraint affecting not only Sortal Classes but also Mixin Classes, therefore it was a constraint affecting Substantial Universals. The specification of (Guizzardi, 2005) mentioned that "every concrete element of [the class diagram] (isAbstract = false) must include in its general collection one class stereotyped as either kind, quantity or collective". However, there is also the 'Mixin Class is abstract' constraint, which claims that all Mixin Classes are abstract. As a consequence, when a Mixin Class was marked as concrete, both the 'Concrete *Substantial* has Substantial Sortal Ancestor' and 'Mixin Class is abstract' constraints were activated. Nevertheless, in this work we claim that when a model has a non-abstract Mixin Class, it is because the user forgot to mark it as abstract, not because he forgot to create the Substance Sortal Ancestor for the Mixin (which is an absurd in the foundational ontology). As a result, concrete *sortals* (not *substantials*) must have a substance sortal ancestor.

We have that "a Category cannot have a RoleMixin as a supertype. In other words, (…) a Category can only be subsumed by another Category or a Mixin". Based on this affirmative, in (Benevides, 2010) there is a constraint specifying that Categories can only specialize Categories or Mixins. The problem with this approach is that all the other constraints about generalization are defined in a negative form, i.e., restricting what classes a certain class *cannot* specialize instead of declaring what classes a certain class *may* specialize. Consequently, Benevides' constraint 'Category specializes Category and Mixin' (positive) together with the constraint 'Mixin Class does not specialize Sortal Class' (negative) would cause an inconvenient validation issue. If a Category specializes one Sortal (viz. Kind, Quantity, Collective, SubKind, Role and Phase) then both of those constraints would be activated. Since those constraints address the same problem, i.e., a single wrong specialization involving a category, we replaced the positive constraint by the negative constraint 'Category does not specialize RoleMixin'. This constraint along with 'Mixin Class does not specialize Sortal Class' cover all the cases of category specialization. With this modification, the only positive constraint about generalizations in our metamodel is 'Object Class specializes Object Class'. This constraint is important to restrict other Classifiers (such as DataTypes, Associations, Moments) from being parents or offspring of Substantial Universals (also called Object Classes in the metamodel).

Some improvements were made in terms of Directed Binary Associations. In Benevides' work, every concrete offspring of Directed Binary Relationship (viz. Mediation, Characterization, Derivation, componentOf, subQuantityOf, subCollectionOf and memberOf) had a constraint specifying that the relation must be binary. For the sake of reuse and simplicity, we claim that there must be an unique constraint 'Directed Binary Association is binary' instead of seven analogous constraints. Similarly, all Dependency Relationship's offspring (viz. Mediation, Characterization and Derivation) had a constraint specifying that the minimum cardinality of the source end (viz. the Relator, the Mode and the Material Association end, respectively) must be greater or equal to 1, and also another constraint specifying that the target end (viz. the Mediated, the Characterized and the Relator end, respectively) must be read only. Again, we claim that there should be only two constraints, one about the Dependency Relationship's source and another about the Dependency Relationship's target. Moreover, since we are dealing with standard UML Associations in our version of the metamodel, many operations were defined to abstract the properties of Directed Binary Associations. With those operations, it is possible to deal with concepts such as source end and target end; part end and whole end (in the case of Meronymics); relator end and mediated end (in the case of Mediation); instead of simply dealing with the concept of member ends as an ordered sequence. Also, those operations could be useful for any possible future applications that must read an OntoUML model via Java code (for instance, a model transformation).

Furthermore, Guizzardi explicitly says that "every mode must be (directly or indirectly) connected to an association end of at least one characterization relation" and "every relator must be (directly or indirectly) connected to an association end of at least one mediation relation". However, about Roles and RoleMixins its simply said that they "must be connected to an association end of a mediation relation". As a result, Benevides implements these statements as a direct or indirect connection in the case of Roles and a direct connection in the case of RoleMixins. However, in our work, we claim that both Role and RoleMixins must be connected *directly or indirectly* to a mediation.

For the sake of modularization, we created operations concerning the "nature" of a Mixin Class. In Benevides' work, although operations concerning the "nature" of a Sortal were present, the operations involving Mixins were directly implemented in the operations involving the "nature" of Relata.

The next modification is related to the memberOf relation with essential parthood. To validate this situation, we need much more than just saying that an essential parthood implies that the whole is a collective and is extensional, like done in (Benevides, 2010). The whole of a memberOf relation is not necessarily a collective, it is an element with collective "nature". Thus, if the whole is a

Mixin Class, then all its collective offspring must be extensional. Otherwise, if the whole is a Sortal, then the collective ancestor (or itself, if the sortal is a collective) must be extensional.

7 CONCLUSIONS

Throughout this work, we defined various elements that belong to a modeling infrastructure for editing and validating OntoUML models. In this final chapter, we summarize all the contributions of this work and mention possible future works.

7.1 CONTRIBUTIONS

In theoretical aspects, we reviewed the OntoUML metamodel specified in (Guizzardi, 2005) in order to comply the UML 2.0 Profile mechanism. This is an important contribution since one of the goals of the OntoUML language is to be a UML lightweight extension. With our review, OntoUML can be specified in any Profile-based tool and OntoUML models can be interchanged between tools as a legitimate UML Profile. Hence, our review not only contributes to the theoretical consistency of the language but impacts its practical application.

Along with the metamodel review, we improved the syntactical constraints defined in (Guizzardi, 2005). For example, in some cases, constraints were overlapping each other, which means the infringement of one of them would automatically cause the infringement of the other. As a consequence, when such constraints were implemented in a validation platform, one particular user mistake would cause more than one constraint violation warning. Therefore, to solve this issue, we defined some constraints in a new way to avoid such overlaps. Additionally, in other cases, we enforced stronger statements than the original ones defined by the author, making constraints more compromised with reality. In terms of implementation, we made improvements when comparing our OCL constraints with the ones provided in (Benevides, 2010). First of all, some of our constraints are more efficient because we made a better use of the OCL language and took better implementation decisions. In some situations, the constraints are simpler (and more efficient) as a consequence of our review of the OntoUML metamodel. Moreover, some of our constraints have a better placement, claiming modularization. As a result, constraints that appear in many places in Benevides' work are concentrated in a single place in our work. Another improvement of our work is some auxiliary methods which may assist users of model manipulations to interact with the OntoUML language in a higher level. Finally, our OntoUML constraints are presented as a reference summary, as opposed to (Guizzardi, 2005) and (Benevides, 2010) in which the constraints are surrounded with theoretical discussions. Thus, the Chapter 6 of this work can serve as a guide for easy consult.

In terms of infrastructure, we addressed the undesirable mixture between concrete and abstract syntax in the metamodel that assists the GMF Editor of (Benevides, 2010). Such metamodel actually represented the center of the infrastructure that preceded our work. Since we argued throughout this work that attaching model manipulations to a metamodel that is specific of a tool or

technology (in this case, GMF) is not very appropriate, we created in response a modeling infrastructure for OntoUML.

One of the main contributions of our infrastructure is the reference metamodel which is a back-end metamodel that is free from any concrete syntax elements and contains all the constraints necessary to validate the syntax of OntoUML models. Just like the UML Profile defined in this work, it complies with the rules of lightweight extension. Additionally, our reference metamodel incorporates the elements of the UML class diagram without any simplification, hence, it is in total accordance with the OMG specification, as opposed to the metamodel implementation of Benevides.

Thus, by defining a reference metamodel with pure abstract syntax and with all the constraints, we open the possibility for existing as many front end tools as necessary. Once a model has been edited and written in terms of a possibly unclean and simplified metamodel, it can easily be transformed into an instance of the reference metamodel, inheriting the already mentioned benefits. In the previous work, constraints are located in the context of the graphical editor's implementation and, therefore, constraints may not be reused by other front end tools. Also, having the GMF editor's metamodel is not sufficient for performing syntax validation, since such process only occurs in the context of the GMF Editor. With our approach, front end tools may now implement validation constraints partially (for example, implement live validation and skip batch validation) and delegate the validation of the complementary constraints to the reference model.

Moreover, because model transformations (such as OntoUML to Alloy and OntoUML to SBVR) and other kinds of model manipulation generally do not concern aspects of model editing and concrete syntax, the reference metamodel is the ideal artifact for performing such actions. Thus, any kind of transformation and manipulation involving the reference metamodel is guaranteed to be independent of any specific front end tool technology. As a matter of fact, our modeling infrastructure was already used to assist a related work that transforms OntoUML models into Structured English (following the rules of OMG's SBVR).

In terms of front end tools, we provided a second alternative for OntoUML graphical editing using the Eclipse tool. For doing so, we specified a UML Profile, following our theoretical revision of the OntoUML metamodel, and obtained a way to apply such Profile to standard UML class diagrams. By doing this, we increase the options users have for editing OntoUML models, since they are able to choose the editor that they find more convenient. Also, by using the UML Profile mechanism to implement our editor, instead of GMF (Benevides, 2010), we are investing in a new technology that may improve in the future. This happens because the profile-based graphical editor of Eclipse is of general interest, since it fulfills the demands of any UML Profile language, not just OntoUML. Therefore, we believe that the usability of the graphical tool tends to be improved by the Eclipse Community over time. In contrast, the GMF Editor can only be improved by our own research group,

since it is specific for the OntoUML language. Finally, the specification of the UML Profile in Eclipse helped us to benchmark our revision of the OntoUML metamodel.

At last, we provided model transformations to convert the models written in any of the graphical editors (the Front End Editor of this work or the GMF Editor of Benevides) to instances of the reference metamodel. This way, we consolidate model editing and model manipulation without the issues of the previous infrastructure.

In summary, the contributions of this work are (from theoretical to practical, and in order of importance):

- A review of the OntoUML metamodel specification; adjusting the OntoUML language to the UML lightweight extension mechanism and, thus, making it compatible with profile-based tools;
- A review of the OntoUML constraints; adding more semantic compromise, eliminating constraint overlap, improving efficiency/modularization, assisting users of model manipulations;
- A specification of a modeling infrastructure for OntoUML; solving the problems of the previous infrastructure related to concrete/abstract syntax and model editing/manipulation;
- A reference metamodel for OntoUML; which is UML 2.0 fully-compliant, accords to the lightweight extension mechanism, possess the complete implementation of the OntoUML constraints in OCL and, hence, is the center of model manipulation in our infrastructure;
- A UML 2.0 compliant profile for OntoUML; benchmarking our review of the OntoUML metamodel and assisting the implementation of our Front End Editor;
- A UML Profile-based Front End Editor; providing a second alternative for graphical editing and benchmarking our infrastructure;
- A transformation from the UML profile defined in this work to the Reference metamodel; integrating model editing and manipulation;
- A transformation from the former GMF Editor's metamodel to the Reference metamodel; integrating the previous work with our work for compatibility reasons.

7.2 FUTURE WORKS

Although we provided a graphical tool in this work, we did not include any kind of validation in it. Therefore, several mistakes committed by the user that could be warned by means of live validation are currently postponed to batch validation in the reference metamodel. The reason for that is the lack of tutorials, documentation and support for the Eclipse UML Profile mechanism. As far as we know, the current version of this tool gives support for batch validation incorporated into the

generated graphical editor, but it does not provide the live validation mechanism implemented in the GMF Editor of (Benevides, 2010).

As a consequence, it is already possible to implement some constraints in the front end metamodel, but some mapping between those constraints and the constraints defined in the reference metamodel is required. This happens because defining constraints in a UML Profile (which deals with UML Metaclasses, Stereotypes and Extensions) is different from specifying constraints in an Ecore metamodel (which basically deals with EClasses) or in a pure UML metamodel. In OntoUML, we want to apply constraints mostly on Stereotypes, not on the existing UML Metaclasses. Thus, some adaptation in the constraints will be required in a possible future work.

Not only live validation but other aspects of usability of our front end could be improved, as soon as new versions of the Eclipse UML Profile mechanism arise. For example, the GMF Editor derives some information about cardinalities of Associations during model editing. Moreover, another toolset, named Papyrus[6] will be incorporated into the Eclipse Model Development Tools[7] (MDT) which is the same project that contains the mechanism we used for constructing our Front End Editor. Papyrus is a dedicated tool for modeling within UML2 and has a variety of facilities for constructing UML Profiles. Therefore, another option is to construct new front end tools based on new technologies that appear (reusing the UML Profile defined here).

[6] http://www.eclipse.org/modeling/mdt/papyrus/ and http://www.papyrusuml.org/
[7] http://www.eclipse.org/modeling/mdt/

REFERENCES

Alloy Community. 2009. http://alloy.mit.edu/community/. [Online] 2009. [Cited: May 31, 2010.]

Benevides, A. 2010. *A Model-Based graphical editor for supporting the creation, verification and validation of OntoUML conceptual models.* Vitória, Brazil : Universidade Federal do Espírito Santo, 2010.

Boldt, Nick and Paternostro, Marcelo. 2006. CASCON'06 Workshop Introduction to the Eclipse Modeling Framework. [Online] 2006. [Cited: June 1, 2010.] http://www.eclipse.org/modeling/emf/docs/presentations/CASCON/CASCON_2006.pdf.

Braga, B. F. B., et al. 2010. Transforming OntoUML into Alloy: towards conceptual model validation using a lightweight formal method. *Innovations in Systems and Software Engineering.* 2010, Vol. 6, pp. 1-13.

Damus, Christian. 2007. Implementing Model Integrity in EMF with MDT OCL. *Eclipse Corner Articles.* [Online] IBM, 2007. [Cited: May 31, 2010.] http://www.eclipse.org/articles/article.php?file=Article-EMF-Codegen-with-OCL/index.html.

Guizzardi, G. 2005. *Ontological Foundations for Structural Conceptual Models.* Enschede, The Netherlands : University of Twente, 2005.

Mulligan, Kevin and Smith, Barry. 1986. A Relational Theory of the Act. *Topoi (5/2), 115-30.* 1986.

Mylopoulos, John. 1992. Conceptual Modelling and Telos. [book auth.] L Loucopoulos and R Zicari. *Conceptual Modelling, Databases and CASE: An Integrated View of Information Systems Development.* New York : Wiley, 1992.

—. 1998. Information Modeling in the Time of the Revolution. *Information Systems.* June 1998, Vol. 23, 3-4.

OMG. 2003. *MDA Guide Version 1.0.* 2003.

—. 2006. *Object Constraint Language.* Version 2.0. 2006.

—. 2009. *OMG Unified Modeling Language (OMG UML), Superstructure.* Version 2.2. 2009.

—. 2008. *Semantics of Business Vocabulary and Business Rules.* Version 1.0. 2008.

Steinberg, Dave. 2008. Fundamentals of the Eclipse Modeling Framework. [Online] March 17, 2008. [Cited: July 2010, 1.] http://www.eclipse.org/modeling/emf/docs/presentations/EclipseCon/EclipseCon2008_309T_Fundamentals_of_EMF.pdf.

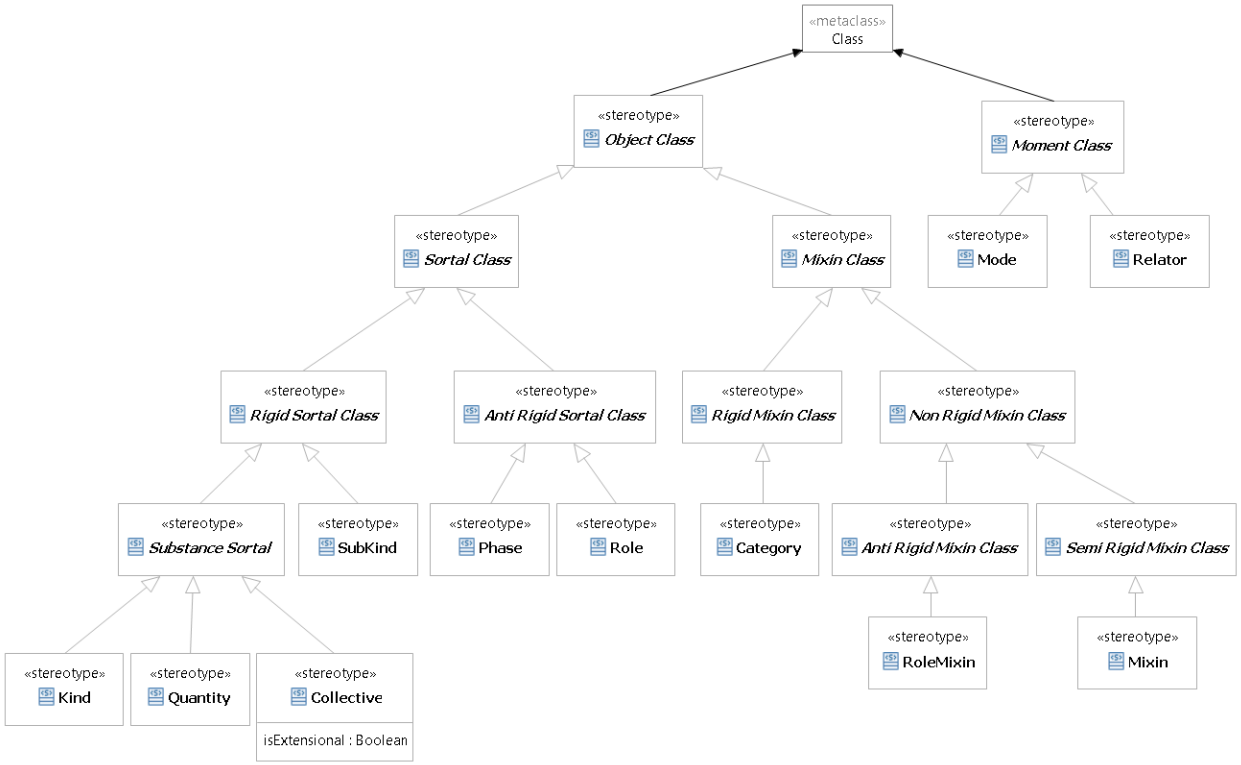
Steinberg, Dave, et al. 2009. *EMF Eclipse Modeling Framework.* Boston : Pearson Education, 2009.

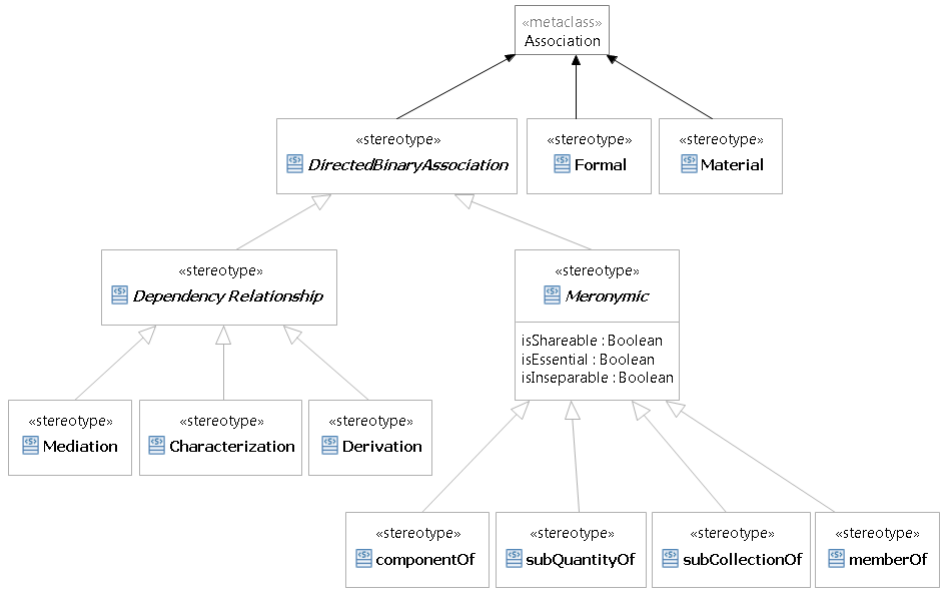
The Eclipse Foundation. 2010. Eclipse Modeling Framework Project (EMF). [Online] 2010. [Cited: May 31, 2010.] http://www.eclipse.org/modeling/emf/.

—. 2010. Graphical Modeling Framework (GMF). [Online] 2010. [Cited: May 31, 2010.] http://www.eclipse.org/modeling/gmp/.

—. 2006. Introduction to UML2 Profiles. [Online] 2006. [Cited: May 31, 2010.] http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Introduction_to_UML2_Profiles/article.html.

APPENDIX A – THE ONTOUML PROFILE





APPENDIX B – THE ONTOUML REFERENCE METAMODEL