Universidade Federal do Espírito Santo

Centro Tecnológico

Programa de Pós-Graduação em Informática

Alessander Botti Benevides

A Model-based Graphical Editor for Supporting the Creation, Verification and Validation of OntoUML Conceptual Models

Vitória - ES, Brazil

February 5th, 2010

Alessander Botti Benevides

A Model-based Graphical Editor for Supporting the Creation, Verification and Validation of OntoUML Conceptual Models

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal do Espírito Santo para obtenção do título de Mestre em Informática.

Orientador:
Giancarlo Guizzardi

Co-orientador:
João Paulo Andrade Almeida

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA
CENTRO TECNOLÓGICO
UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

Vitória - ES, Brazil

February 5th, 2010

Dados Internacionais de Catalogação-na-publicação (CIP)
(Biblioteca Central da Universidade Federal do Espírito Santo, ES, Brasil)

Benevides, Alessander Botti, 1982-
B465m A model-based graphical editor for supporting the creation, verification and validation of OntoUML conceptual models / Alessander Botti Benevides. – 2010.
289 f. : il.

Orientador: Giancarlo Guizzardi.
Co-Orientador: João Paulo Andrade Almeida.
Dissertação (Mestrado em Informática) – Universidade Federal do Espírito Santo, Centro Tecnológico.

1. Ontologia. 2. Lógica. 3. Modelagem conceitual. 4. Verificação de modelos conceituais. 5. Validação de modelos conceituais. I. Guizzardi, Giancarlo. II. Almeida, João Paulo Andrade. III. Universidade Federal do Espírito Santo. Centro Tecnológico. IV. Título.

CDU: 004

Dissertação de Mestrado sob o título *"A Model-based Graphical Editor for Supporting the Creation, Verification and Validation of OntoUML Conceptual Models"*, defendida por Alessander Botti Benevides e aprovada em 5 de Fevereiro, 2010, em Vitória, Estado do Espírito Santo, pela banca examinadora constituída pelos doutores:

Prof. Dr. Giancarlo Guizzardi
Departamento de Informática - UFES
Orientador

Prof. Dr. João Paulo Andrade Almeida
Departamento de Informática - UFES
Co-orientador

Prof. Dr. Ricardo de Almeida Falbo
Departamento de Informática - UFES
Examinador Interno

Prof. Dr. Mateus Conrad Barcellos da Costa
Coordenadoria de Informática - Instituto
Federal do Espírito Santo (Ifes) - Campus Serra
Examinador Externo

To all beings, with great respect and love.

Acknowledgements

I thank my parents (Cecilia and Adwalter) for teaching me the value of perseverance, my brother (Alessandro) for his friendship, and my fiancée (Renata) for her love and patience during these years of dedication to this research. I thank all my family and friends for the support and love. Without their support, this thesis would not be possible.

I also specially thank my advisors Giancarlo and João Paulo for their assistance, friendship and comprehension.

Gian, I thank you very much for showing me a whole new world of the applications of philosophy in computer science. Before I met you, I had no idea that I could conciliate my passion for philosophy with my career as a B.Eng. in Computer Engineering. Thank you very much!

João Paulo, I thank you very much for your technical suggestions and help, as well as for your comprehension and helpfulness in providing me scholarships!

Many thanks to Bernardo Ferreira Bastos Braga for his great contribution on this research, by his ideas, suggestions and corrections.

I also thank Bernardo Nunes Gonçalves for his friendship and motivation and Kyriakos Anastasakis for his support and suggestions.

Moreover, this research has been supported by Conselho Nacional de Desenvolvimento Científico e Tecnológico[1] (CNPq), Fundação de Apoio à Ciência e Tecnologia do Espírito Santo[2] (FAPES) (INFRA-MODELA) and Fundo de Apoio à Ciência e Tecnologia do Município de Vitória[3] (FACITEC) (MODELA). I thank very much these organizations.

Finally, I thank you for the interest in my thesis!

[1] http://www.cnpq.br.

[2] http://www.fapes.es.gov.br.

[3] http://www.vitoria.es.gov.br/secretarias/sedec/facitec.htm.

"You must be the change you want to see in the world."

"We should be able to refuse to live if the price of living be the torture of sentient beings."

Mohandas Karamchand Gandhi

Resumo

Esta dissertação de mestrado apresenta um editor gráfico baseado em modelos para o suporte à criação, verificação e validação de modelos conceituais e ontologias de domínio em uma linguagem de modelagem filosoficamente e cognitivamente bem-fundada chamada OntoUML. O editor é projetado para proteger o usuário da complexidade dos princípios ontológicos subjacentes a essa linguagem. Adicionalmente, o editor garante a aplicação destes princípios nos modelos produzidos por prover um mecanismo para verificação formal automática de restrições, daí assegurando que os modelos criados serão sintaticamente corretos.

Avaliar a qualidade de modelos conceituais é um ponto chave para assegurar que os mesmos podem ser utilizados efetivamente como uma base para o entendimento, consentimento e construção de sistemas de informação. Por essa razão, o editor é também capaz de gerar instâncias de modelos automaticamente por meio da transformação desses modelos em especificações na linguagem Alloy. Como as especificações Alloy geradas incluem os axiomas modais da ontologia de fundamentação subjacente à OntoUML, chamada Unified Foundational Ontology (UFO), as instâncias geradas automaticamente vão apresentar um comportamento modal enquanto estiverem sendo classificadas dinamicamente, suportando, assim, a validação das meta-propriedades modais dos tipos fornecidos pela linguagem OntoUML.

Abstract

This thesis presents a model-based graphical editor for supporting the creation, verification and validation of conceptual models and domain ontologies in a philosophically and cognitively well-founded modeling language named OntoUML. The editor is designed in a way that, on one hand, it shields the user from the complexity of the ontological principles underlying this language. On the other hand, it reinforces these principles in the produced models by providing a mechanism for automatic formal constraint verification, hence ensuring that the created models will be syntactically correct.

Assessing the quality of conceptual models is key to ensure that conceptual models can be used effectively as a basis for understanding, agreement and construction of information systems. For this reason, the editor is also capable of automatic generation of model instances by transforming these models into specifications in the logic-based language Alloy. As the generated Alloy specifications include the modal axioms of the foundational ontology underlying OntoUML, named Unified Foundational Ontology (UFO), then the automatically generated instances will present modal behaviour while being dynamically classified, thereby supporting the validation of the modal meta-properties of the OntoUML types.

List of Figures

List of Listings

List of Definitions

List of Acronyms

ADT	ATL Development Tools[1]
AEON	Automatic Evaluation of ONtologies[2]
AIX	Advanced Interactive eXecutive[3]
ALU	Arithmetic and Logic Unit
AMMA	ATLAS Model Management Architecture[4] (Bézivin *et al.*(1), 2004 apud Jouault *et al.*(2), 2006, p. 1)
API	Application Programming Interface
ASSL	A Snapshot Sequence Language
ATL	ATLAS Transformation Language[5]
ATLAS	ATLantic dAta Systems (INRIA and LINA)[6]
CASE	Computer-Aided Software Engineering
CNPq	Conselho Nacional de Desenvolvimento Científico e Tecnológico[7]
CPU	Central Process Unit
DOE	Differential Ontology Editor[8]
DSL	Domain-Specific Language
EER	Enhanced Entity-Relationship

[1] http://www.eclipse.org/m2m/atl.

[2] http://ontoware.org/projects/aeon.

[3] http://www-03.ibm.com/systems/power/software/aix.

[4] http://atlanmod.emn.fr.

[5] http://www.eclipse.org/m2m/atl.

[6] http://atlanmod.emn.fr/AMMA/atlas.

[7] http://www.cnpq.br.

[8] http://homepages.cwi.nl/~troncy/DOE.

EMF	Eclipse Modeling Framework[9] (3)
EMOF	Essential MOF (4)
EPLv1	Eclipse Public License - v1.0[10]
FAPES	Fundação de Apoio à Ciência e Tecnologia do Espírito Santo[11]
FACITEC	Fundo de Apoio à Ciência e Tecnologia do Município de Vitória[12]
FOL	First Order Logic
FOSS	Free and Open Source Software
F-Logic	Frame Logic (5)
GEF	Graphical Editing Framework[13] (3)
GIMP	GNU Image Manipulation Program[14]
GTK	GIMP Toolkit[15]
GMF	Graphical Modeling Framework[16]
GPLv3	GNU General Public License Version 3[17]
GNU	GNU's Not Unix![18]
HP-UX	Hewlett Packard UniX[19]
IBM	International Business Machines[20]
IDE	Integrated Development Environment
iff	if and only if

[9] http://www.eclipse.org/modeling/emf.

[10] http://www.eclipse.org/org/documents/epl-v10.html.

[11] http://www.fapes.es.gov.br.

[12] http://www.vitoria.es.gov.br/secretarias/sedec/facitec.htm.

[13] http://www.eclipse.org/gef.

[14] http://www.gimp.org.

[15] http://www.gtk.org.

[16] http://www.eclipse.org/modeling/gmf.

[17] http://www.fsf.org/licensing/licenses/gpl.html.

[18] http://www.gnu.org.

[19] http://www.hp.com/go/hpux.

[20] http://www.ibm.com.

INRIA	Institut national de recherche en informatique et en automatique[21]
JET	Java Emitter Templates[22]
JRE	Java Runtime Environment
KM3	Kernel Meta Meta Model[23] (6)
LINA	Laboratoire d'Informatique de Nantes Atlantique[24]
MDA	Model-Driven Architecture[25] (7)
MDE	Model-Driven Engineering
MDT	Model Development Tools[26]
MOF	Meta-Object Facility[27] (4)
MVC	Model-View-Controller
OCL	Object Constraint Language (8)
OMG	Object Management Group[28]
OML	OPEN Modelling Language (9)
OPEN	Object-oriented Process, Environment and Notation[29]
OS	Operational System
OS/360	IBM System/360 Operating System
OWL	Web Ontology Language (10)
PIM	Platform-Independent Model[30] (7)
PDM	Platform Definition Model[31] (7)

[21] http://www.inria.fr.

[22] http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html.

[23] http://wiki.eclipse.org/KM3.

[24] http://www.lina.univ-nantes.fr.

[25] http://www.omg.org/mda.

[26] http://www.eclipse.org/modeling/mdt.

[27] http://www.omg.org/mof.

[28] http://www.omg.org.

[29] http://www.open.org.au.

[30] http://www.omg.org/mda.

[31] http://www.omg.org/mda.

PowerPC	Performance Optimization With Enhanced RISC - Performance Computing
PSM	Platform-Specific Model[32] (7)
RISC	Reduced Instruction Set Computing
SAT	Boolean satisfiability problem
SPARC	Scalable Processor Architecture[33]
SUMO	Suggested Upper Merged Ontology[34]
SUO	Standard Upper Ontology[35]
UFO	Unified Foundational Ontology (11)
UML	Unified Modeling Language[36] (12, 13)
URL	Uniform Resource Locator
USE	UML Specification Environment[37] (14)
wff	well-formed formula
WPF	Windows Presentation Foundation[38]
XMI	XML Metadata Interchange (15)
XML	eXtensible Markup Language[39]

[32] http://www.omg.org/mda.

[33] http://www.sparc.org.

[34] http://www.ontologyportal.org.

[35] http://suo.ieee.org.

[36] http://www.uml.org.

[37] http://www.db.informatik.uni-bremen.de/projects/USE.

[38] http://windowsclient.net/wpf.

[39] http://www.w3.org/XML.

List of Symbols

$\neg$	negation logical operator
$\wedge$	logical conjunction operator
$\vee$	logical disjunction operator
$\oplus$	logical exclusive disjunction operator
$\rightarrow$	conditional logical operator or function arrow, depending on the context
$\leftrightarrow$	biconditional logical operator
$\exists$	existential quantification operator
$\exists!$	uniqueness quantification operator
$\forall$	universal quantification operator
$\Box$	necessity logical operator
$\Diamond$	possibility logical operator
$\triangleq$	definition operator
$\blacksquare$	end of proof symbol
$::$	instantiation operator
$\imath$	definite description operator
$=$	equality operator
$\neq$	difference operator
$+$	addition operator between natural numbers
$-$	negative operator for natural numbers, or minus operator between natural numbers, or set-theoretic complement between sets, depending on the context
$\times$	multiplication operator between natural numbers or Cartesian product between two sets, depending on the context
$>$	greater than operator between natural numbers
$\geq$	greater or equal than operator between natural numbers
$<$	less than operator between natural numbers or proper parthood, depending on the context

$\leq$	less or equal than operator between natural numbers or parthood, depending on the context
$\sharp$	cardinality operator
Σ	summation operator between natural numbers
$\emptyset$	the empty set
$\in$	set-theoretic membership operator
$\bigcup$	set-theoretic union operator
$\cap$	set-theoretic intersection operator
$\mathbb{N}$	set of all natural numbers
$\mathbb{N}^*$	set of all natural numbers excluding the zero ($\mathbb{N} - \{0\}$)
$\aleph_0$	the smallest infinite cardinal number
$*$	used to represent $\aleph_0$ cardinalities in conceptual models
i	inherence relation
β	Bearer of a Moment
m	mediation relation
L	a language of quantified modal logics with identity
M	model structure for the language L: $\langle \mathcal{K}, \mathcal{D}, v \rangle$
$\mathcal{W}$	set of all worlds
R	binary accessibility relation defined in $\mathcal{W} \times \mathcal{W}$
$\mathcal{K}$	Kripke structure
$\mathcal{D}$	*possibilist* domain of quantification
$\mathcal{D}(w)$	*actualist* domain of quantification regarding the world w
ε	if the domain of quantification is a *possibilist* one, then ε is a predicate that states the existence of an individual $x \in \mathcal{D}$ within a world $w \in \mathcal{W}$; if the domain of quantification is an *actualist* one, then ε is a predicate regarding the pertinence of an individual x within $\mathcal{D}(w)$ and can be explicitly defined such that $\varepsilon(x) \triangleq \exists y(y = x)$
δ	interpretation function that assigns values to the non-logical constants of the language L and a model structure M

Contents

1 Introduction

This master thesis presents the results of our research on building Model-Driven Engineering (MDE) based tools for supporting the activities of construction, verification and validation of conceptual models, particularly OntoUML conceptual models or ontologies.

The aim of this chapter is to introduce our motivation for this research (section 1.1), our goals and scope (section 1.2), our approach (section 1.3) and the structure of the thesis (section 1.4).

1.1 Motivation

In 16, Mylopoulos defines conceptual modeling as “the activity of formally describing some aspects of the physical and social world around us for purposes of understanding and communication”. In this view, a conceptual model is a means to represent what modelers (or stakeholders represented by modelers) perceive in some portion of the physical and social world, *i.e.*, a means to express their conceptualization (17) of a certain universe of discourse.

Furthermore, Guizzardi defines ontology, in Computer Science, as “a conceptual specification that describes knowledge about a domain in a manner that is independent of epistemic states and state of affairs. Moreover, it intends to constrain the possible interpretations of a language’s vocabulary so that its logical models approximate as well as possible the set of intended world structures of a conceptualization *C* of that domain.” (17). In other words, “An ontology can be seem as the metamodel specification for an ideal language to represent phenomena in a given domain in reality, *i.e.*, a language which only admits specifications representing possible state of affairs in reality.” (17). Therefore, we understand an ontology as a conceptual model that is homomorphic to an ideal conceptualization of the domain.

We believe that, if conceptual models are to be used effectively as a basis for understanding, agreement, and, perhaps, construction of an information system, conceptual models should express as accurately as possible a modeler’s intended conceptualization. More specifically, the model should ideally describe all states of affairs that are deemed *admissible* and rule out those deemed *inadmissible* according to the conceptualization (17).

In pace with Degen *et al.*, we argue that "every domain-specific ontology must use as framework some upper-level ontology" (18). This claim for an upper-level (or foundational) ontology underlying a domain-specific ontology is based on the need for fundamental ontological structures, such as theory of parts, theory of wholes, types and instantiation, identity, dependence, unity, *etc.*, in order to properly represent reality. From an ontology representation language perspective, this principle advocates that, for a modeling language to meet the requirements of expressiveness, clarity and truthfulness in representing the subject domain at hand, it must be an ontologically well-founded language in a strong ontological sense, *i.e.*, it must be a language whose modeling primitives are derived from a proper foundational ontology (19, 17).

An example of a general conceptual modeling and ontology representation language that has been designed following these principles is the Unified Modeling Language[1] (12, 13) (UML) profile proposed in 11. This language (later termed OntoUML) has been constructed in a manner that its metamodel reflects the ontological distinctions prescribed by Unified Foundational Ontology (11) (UFO). UFO is a foundational ontology designed specially for conceptual modeling languages. The ontological categories comprising UFO are motivated by a number of theories in formal ontology, philosophical logics, cognitive science and linguistics. Moreover, formal constraints have been incorporated in this language's metamodel in order to incorporate the formal axiomatization in UFO. Therefore a UML model that is ontologically misconceived taking UFO into account is syntactically invalid when written in OntoUML.

However, one would certainly be naive to assume that modelers make no mistakes while constructing the models and that they fully understand the theory that supports the language. These cases could lead to ill-defined conceptual models, which may be: (i) syntactically incorrect; (ii) syntactically correct, but unsatisfiable; (iii) syntactically correct, satisfiable, but invalid according to the intended conceptualization.

Although OntoUML has been able to provide mechanisms for addressing a number of classical conceptual modeling problems (20), and the language has been successfully employed in application domains (21), (22), there was still no tool support for building, verifying (syntax checking) or validating conceptual models and domain ontologies constructed using OntoUML. Therefore, the aim of this thesis is to build a graphical editor in which one could build, verify and validate an OntoUML model.

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

1.2 Goals and Scope

We aim at designing an editor in a way that, on one hand, it can shield the user from the complexity of the ontological principles underlying OntoUML, and, on the other hand, it can reinforce these principles in the produced models by providing mechanisms for automatic formal constraint verification, automatic model filling and assisting the validation of the models.

In the definition of the scope of the editor, we understand that it must:

- Allow the creation of conceptual models and ontologies graphically, in a simple way;
- Automatically verify models (*i.e.*, check the OntoUML syntax constraints in models), when suitable;
- Allow the modeler to start syntactic checks manually, when he/she deems suitable;
- Inform the reason why a model is syntactically invalid in a way the modeler understands what is wrong, so he/she can figure out how to fix it;
- Automatically derive information from the models in specific contexts that will be shown later, saving the user from modeling information that could be automatically inferred;
- Allow the generation of model instances with the purpose of improving the modeler's confidence in the validity of the model.

Therefore, the primary contribution of this thesis is to present a model-based OntoUML graphical editor with support for: (i) automatic and manual verification of models; (ii) automatic completion of certain parts of the model, in specific contexts; and (iii) automatic generation of instances for supporting model validation.

1.3 Approach

In order to accomplish our goals of building an editor capable of automatic syntax checking and automatic model filling, we employ Model-Driven Architecture[2] (7) (MDA) technologies. The architecture of the editor has been conceived to follow a Model-Driven Approach. In particular, we have adopted the Object Management Group[3] (OMG) Meta-Object Facility[4] (4) (MOF)

[2] http://www.omg.org/mda.

[3] http://www.omg.org.

[4] http://www.omg.org/mof.

metamodeling architecture, the languages Ecore (23), Object Constraint Language (8) (OCL), the Eclipse platform (24) and some of its plug-ins, mainly Eclipse Modeling Framework[5] (3) (EMF), Model Development Tools[6] (MDT) and Graphical Modeling Framework[7] (GMF).

Furthermore, in order to accomplish our requirement of making possible the automatic generation of model instances we will follow an approach based on the transformation of OntoUML models into formal specifications in the logic-based language Alloy (25, 26).

In our approach, the Alloy specification is provided to the Alloy Analyzer to generate an instance[8] composed of a set of *atoms* and *relations* (27, pp. 35-48) representing instances of the classifiers taken from the OntoUML model and a world structure that reveals the possible dynamics of object creation, classification, association and destruction. Each world in this structure represents a snapshot of the objects and relations that exist in that world. This world structure is necessary since the meta-properties characterizing most of the ontological distinctions in UFO are modal in nature. For example, the definition of a "rigid" classifier states that it applies necessarily to its instances in all worlds in which they exist (see Definition 21). We have specified UFO's modal axioms in Alloy to guarantee that the generated world structure satisfies those axioms by construction. Therefore, the sequence of possible snapshots in this world structure will improve a modeler's confidence on claims of validity.

In general, we will perform the following tasks:

1. Implement the OntoUML abstract syntax (metamodel) (11, p. 316, 334, 348) as a MOF complying metamodel, using the Ecore language, and implement the derivation of the derived meta-relations as OCL expressions within the Ecore metamodel;

2. Define a number of OCL expressions for the automatic initialization or modification of some meta-attributes' values, *e.g.*, OCL expressions for the calculation of the values of the upper cardinality constraint for both association ends of Material Associations and the lower and upper cardinality constraints of the source association end of Derivation relationships[9] (11, p. 331, Figs. 8-10);

3. Define the OntoUML syntactical constraints (11, ps. 317–320, 334–338, 348–352) as OCL expressions;

[5] http://www.eclipse.org/modeling/emf.

[6] http://www.eclipse.org/modeling/mdt.

[7] http://www.eclipse.org/modeling/gmf.

[8] In order to avoid the many overloadings of the term "model" (*e.g.*, the distinct meanings of "model" in Conceptual Modeling and in Model Theory), when referring to models in the sense defined in Model Theory, the Alloy developers call them instances instead (27, p. 267). Informally, in Model Theory, a model for a sentence *S* is an *interpretation I* that makes *S* state something true.

[9] For these OntoUML concepts, see chapter 3.

4. Define the OntoUML concrete syntax (11, ps. 317–320, 334–338, 348–352) by using GMF;

5. Use the EMF, MDT and GMF plug-ins in order to build the editor;

6. Revisit the *possibilist*[10] logical formalization of some UFO concepts, formalizing them in an *actualist*[11] modal logic, occasionally also expanding them in order to adapt the original formal characterization to the proposed objectives of this work. This task is important to enable the automatically generated model instances to show the dynamics of creation and destruction of individuals;

7. Categorize and order the worlds defined for the totally accessible world structure of QS5[12] into different common sense types of temporal worlds ordered by a partial order relation of succession, so we can simulate a (branching time) temporal logic while maintaining the modal neutrality of UFO, which do not point necessarily to a temporal interpretation. While UFO's formalization is based on QS5, having totally accessible world structures, we believe that the inspection of instances in which worlds can the interpreted temporally is more suitable for validation purposes. Thus, we constrain the world structures so that only instances that are consistent with this temporal perspective can be generated;

8. Specify the QS5 world structure with our defined temporal types and ordering in Alloy;

9. Find mapping patterns from OntoUML models to Alloy specifications;

10. Finally, we will create an ATLAS Transformation Language[13] (ATL) automatic transformation in order to transform OntoUML models into Alloy specifications.

Concluding, almost every task described above generates an MDE artifact, which is a secondary contribution of this work. These artifacts are:

(a) An OntoUML metamodel in Ecore, with derived meta-relations implemented as OCL expressions;

(b) A set of OCL expressions formalizing the automatic initialization or modification of some meta-attributes' values, *e.g.*, OCL expressions for the calculation of the values of the upper cardinality constraint for both association ends of Material Associations and the lower and

[10] For possibilism, see section 2.1.

[11] For actualism, see section 2.1.

[12] For the QS5 logics, see section 2.1.

[13] http://www.eclipse.org/m2m/atl.

upper cardinality constraints of the source association end of Derivation relationships (11, p. 331, Figs. 8-10);

(c) A set of OCL expressions formalizing the OntoUML syntactical constraints;

(d) A GMF definition of the OntoUML concrete syntax;

(e) An actualist logical formalization of some UFO concepts;

(f) A categorization of worlds in a common sense temporal structure;

(g) An Alloy specification of the QS5 world structure with our defined temporal types and ordering;

(h) A transformation specification from OntoUML models to Alloy specifications.

A UML activity diagram regarding those tasks and artifacts is shown in Fig. 1.

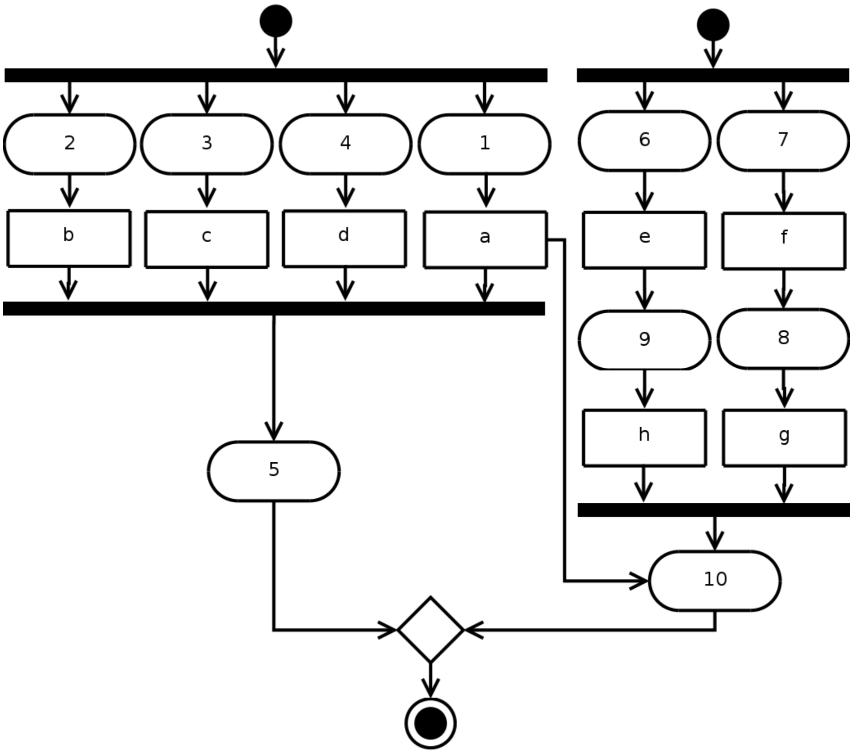


Figure 1: UML activity diagram for the actions that lead to the building of the editor.

1.4 Structure of the Thesis

Besides this introductory chapter, this thesis is organized in five additional chapters. Chapter 2 contains a theoretical background about (i) the system of modal logics employed in this thesis,

(ii) a number of MDE-related technologies that are relevant for this thesis, and (iii) the Alloy language and tools. Chapter 3 contains an introduction on UFO and OntoUML. Chapter 4 is about building a tool for model building and verification. Chapter 5 is about building a tool for supporting model validation. Finally, in chapter 6 we pose our final considerations.

In addition, there are five appendices and two annexes. Appendix A contains the definition of the OntoUML syntactical constraints in OCL. Appendix B contains the ATL implementation of the transformation from OntoUML models to Alloy specifications. Appendix C contains a short guide for installing and using the OntoUML graphical editor. Appendix D contains a manual for installing and using the OntoUML to Alloy transformation. Appendix E contains the themes created in order to customize the instances generated by the Alloy Analyzer. Annex A contains the GNU General Public License Version 3[14] (GPLv3) and annex B contains the Eclipse Public License - v1.0[15] (EPLv1).

This thesis is licensed under 🅭 BY: [16].

[14] http://www.fsf.org/licensing/licenses/gpl.html.

[15] http://www.eclipse.org/org/documents/epl-v10.html.

[16] http://creativecommons.org/licenses/by/3.0/legalcode.

2 Background

In this chapter we present the theoretical background for this thesis. In section 2.1, we briefly comment on the system of modal logics employed in this thesis. Section 2.2 presents a number of MDE-related technologies that are relevant for this thesis, namely, MOF, OCL and ATL (in this order). Moreover, the same subsection also presents the Eclipse Integrated Development Environment (IDE) and a number of its plug-ins (EMF, GMF and MDT), which are important to achieve the purposes of this work. Section 2.3 briefly presents the Alloy language and its analyzer, named Alloy Analyzer.

2.1 Modal Logic

In this thesis, we employ two logic systems in order to formalize our logical expressions and definitions, which are the First Order Logic (FOL) and a Quantified Modal Logics with Identity. In the following, we will present only the latter logic.

Firstly, modal logics deals with the characterization of the *modes* in which a proposition may be true or false, more specifically, their possibility, necessity and impossibility (28, p. 20).

There are many modal semantics for modal logics. We will be interested in a specific one, which employs the notion of possible worlds composing Kripke structures (29) (which are also called world structures). One can intuitively understand possible worlds as state of affairs that are/were possible to happen. For example, future state of affairs are possible to happen (from the present), while counterfactual state of affairs were possible to happen in the past.

Moreover, there are many interpretations regarding the ontological status of possible worlds, such as modal possibilism, actualism, realism, meinongianism, combinatorialism, *etc.* (28, pp. 29-32). However, a full discussion of the topic is outside the scope of this background. Therefore, we will only discuss about the notions of actualism and possibilism, which will be important to us.

Classical possibilism makes an ontological distinction to be drawn between being, on the one hand, and existence, or actuality, on the other. Being is the broader of the two notions, encompassing absolutely everything there is in any sense. For the classical possibilist, every existing thing is, but not everything there is exists. Things that do not exist but could have existed are known as (mere) *possibilia* (30). In a possibilist modal logic, there is a unique domain of quantification $\mathcal{D}$ that is the set of all beings, named *possibilia*. Therefore, in order to state that an individual $x \in \mathcal{D}$ exists in a world $w \in \mathcal{W}$ (where $\mathcal{W}$ is the set of all possbile worlds), one shall explicitly create a predicate $\varepsilon(x)$, which may have different extensions in different worlds.

Contrariwise, actualism does not accept this distinction between being and existence, stating that everything that can in any sense be said to be, exists (in other words, is actual or obtains), and denying that there is any kind of being beyond actual existence. In other words, to be is to exist, and to exist is to be actual (31). Furthermore, an actualist modal logic will have a varying domain of quantification $\mathcal{D}(w)$, because for each world $w \in \mathcal{W}$ there may be a distinct set of individuals $x \in \mathcal{D}(w)$ that exist in w. Therefore, in an actualist system, the existence operator ε can then be explicitly defined such that $\varepsilon(x) \triangleq \exists y(y = x)$.

For the modal propositions created in the present thesis, we make use of a language L of quantified modal logics with identity. The alphabet of L contains the traditional operators of $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation), $\rightarrow$ (conditional), $\leftrightarrow$ (biconditional), $\forall$ (universal quantification), $\exists$ (existential quantification), with the addition of the equality operator =, the uniqueness existential quantification operator $\exists!$, and the modal monadic operators $\Box$ (necessity) and $\Diamond$ (possibility). Regarding these modal operators, if A is a well-formed formula (wff) in FOL than $\Box$A is a wff in this logic and is read as “It is necessarily the case that A” and $\Diamond$A is also a wff in this logic, being read as “It is possibly the case that A”. The following holds for these three latter operators: (1) $\Diamond \mathrm{A} \triangleq \neg\Box\neg \mathrm{A}$; (2) $\Box \mathrm{A} \triangleq \neg\Diamond\neg \mathrm{A}$ and (3) $\exists! x(\mathrm{A}) \triangleq \exists y(\forall x(\mathrm{A} \leftrightarrow (x = y)))$. Additionally, we add that the models assumed here are the so-called *normal models* (32), *i.e.*, the equality operator is defined between individuals in the domain of quantification in each world, and equality if it holds, it holds necessarily. In other words, the formula $\forall x, y((x = y) \rightarrow \Box(x = y))$ is valid.

Now, in order to formalize the semantics of this language, we will make use of Kripke structures. A Kripke structure $\mathcal{K}$ is a $\langle \mathcal{W}, R\rangle$ structure in which $\mathcal{W}$ is a non-empty set of worlds and R is a binary accessibility relation defined in $\mathcal{W} \times \mathcal{W}$. We denote that a world w access a world w' by wRw'.

A Model-Theoretic semantics for this language can be given by defining an interpretation function δ that assigns values to the non-logical constants of the language and a model structure M. In this language M has a structure $\langle \mathcal{K}, \mathcal{D}\rangle$ where $\mathcal{K}$ is a Kripke structure, and $\mathcal{D}$ can be (i) a possibilist domain of quantification comprising a set of beings or (ii) an actualist varying domain of quantification that is a function from worlds to non-empty domains of objects that are assumed to exist in that worlds.

Here, unless explicitly mentioned, we take worlds to represent maximal states of affairs (states of the world). Informally, we can state that the truth of *formulæ* involving the modal operators can be defined such that the semantic value of formula □A is true in world w if and only if (iff) A is true in every world $w^\backprime$ accessible from w. Likewise, the semantic value of formula ◊A is true in world w iff A is true in at least one world $w^\backprime$ accessible from w.

Finally, in chapter 3, following the original formal characterization of UFO and OntoUML language (11), we assume all worlds to be equally accessible and, as a result, we have the language of a possibilist quantified modal logic QS5. However, in order to facilitate the understanding of the dynamics of creation and destruction of OntoUML instances within worlds, in section 5.3 we will revisit the formalization of some UFO concepts in an actualist quantified modal logic QS5 with a varying domain of quantification, so we do not have to define an explicit existence predicate, which will be identified with the varying domain of quantification.

2.2 MDE Technologies

Since the creation of the first programming languages, software researchers were concerned with the need of creating abstractions in order to help software developers to program in terms of their design intent rather than the underlying computing environment, *e.g.*, Central Process Unit (CPU), memory, *etc.*. So, the programming languages should shield programmers from the complexities of these environments.

For example, early programming languages, such as assembly and Fortran, shielded developers from complexities of programming with machine code. Likewise, early operating system platforms, such as IBM System/360 Operating System (OS/360) and Unix, shielded developers from complexities of programming directly to hardware (33, p. 25).

Although these early languages and platforms raised the level of abstraction, they still had a distinct “computing-oriented” focus. In particular, they provided abstractions of the solution space - that is, the domain of computing technologies themselves - rather than abstractions of the problem space that express designs in terms of concepts in application domains, such as telecom, aerospace, healthcare, insurance, and biology (33, p. 25).

Since the emergence of the first Computer-Aided Software Engineering (CASE) tools, researchers are trying to transform models into code, but the limitations of the modeling languages, coding languages and platforms available at the time limited the use of these tools in software development.

Advances in languages and platforms during the past two decades have raised the level of software abstractions available to developers. For example, the reuse of class libraries in Object-oriented languages, such as C++ or Java; the creation of Domain-Specific Languages (DSLs), which offer less general primitives than the ones of general-purpose modeling languages (such as UML) (34, p. 7) allowing solutions to be expressed in the idiom and at the level of abstraction of the problem domain; and the advances in generic programming (35, 36).

Now, the field of MDE aims at consider models as first class entities, providing standards for metamodeling, such as the MDA (subsection 2.2.1) and MOF and OCL languages (see subsections 2.2.2 and 2.2.3, respectively), for model transformation, such as the ATL language (see subsection 2.2.4), as well as tools, such as the Eclipse (subsection 2.2.5) and some of its plug-ins, like EMF (subsubsection 2.2.5.1), GMF (subsubsection 2.2.5.2) and MDT (subsection 2.2.5).

2.2.1 MDA

The best known MDE initiative is the MDA, from the Object Management Group[1] (OMG)[2]. This approach defines system functionality using a Platform-Independent Model[3] (7) (PIM) by means of an appropriate DSL. Then, given a Platform Definition Model[4] (7) (PDM), the PIM is translated to one or more Platform-Specific Models[5] (7) (PSMs).

The MDA is related to a number of standards, such as UML, OCL, MOF and XML Metadata Interchange (15) (XMI). In the subsections 2.2.2 and 2.2.3 we make a brief introduction to MOF and to OCL, respectively.

Also, some MDA standards are implemented in a tool named Eclipse, which we make extensive use in the present thesis. Eclipse is briefly presented in subsection 2.2.5.

2.2.2 MOF

MOF concretely defines a subset of UML for describing class modeling concepts within an object repository. MOF was first standardized in 1997, at the same time as UML. The standard

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

[2] http://www.omg.org

[3] http://www.omg.org/mda.

[4] http://www.omg.org/mda.

[5] http://www.omg.org/mda.

is available at 4 (37, pp. 39-40).

MOF is a meta-meta-model that is self-defined by using a reflexive definition. It is based mainly on three concepts, namely entity, association and package, in addition with a set of primitive types. Furthermore, MDA postulates the use of MOF as the unique meta-meta-model for writing meta-models (38).

In order to write metamodels within the EMF plug-in of Eclipse, we have used the Ecore language, which is very similar to the Essential MOF (4) (EMOF) language (see section 2.2.5.1).

2.2.3 OCL

We used the OCL language in order to formalize the syntactical constraints of OntoUML. One of the reasons we chose OCL was the tool support (*e.g.* parsers and interpreters) for this language. By using OCL, the construction of the mechanism for automatic model verification was straightforward.

The purpose of OCL is to complement the UML language. A UML diagram, such as a class diagram, is typically not refined enough to provide all the relevant aspects of a specification. Often, there is a need to describe additional constraints about the objects in the model, and such constraints are frequently described in natural language, which can result in ambiguities. In order to write unambiguous constraints, so-called formal languages, as Object Constraint Language (8) (OCL), have been developed (8, p. 5).

OCL has been developed as a business modeling language within the International Business Machines[6] (IBM) Insurance division, and has its roots in the Syntropy method (39) (8, p. 5), being based on a set theory and predicate logics and having a formal mathematical semantics (40).

OCL is a pure specification language. Therefore, an OCL expression is guaranteed to be without side effects[7](8, p. 5). As a declarative language, OCL expressions specify what is a valid instance for a given UML/MOF (meta)model and not how an instance can be created (like procedural languages do). Therefore, by using OCL, the modeler can describe a domain abstracting the implementation's issues.

Also, OCL is a typed language, *i.e.*, each OCL expression has a type (8, p. 5). This makes

[6] http://www.ibm.com.

[7] "When an OCL expression is evaluated, it simply returns a value. It cannot change anything in the model. This means that the state of the system will never change because of the evaluation of an OCL expression, even though an OCL expression can be used to specify a state change (*e.g.*, in a post-condition)"(8, p. 5).

possible the checking of expressions before building an executable version of the model, so one can detect errors in initial stages of the modeling process.

In UML 1.X, OCL was a language utilized in order to express constraints in model diagrams. This means that, although the diagrams indicate that certain objects or values could be present in the modeled system, these values are only valid if the constraints specified by the OCL invariants were satisfied (40).

In UML 2.0, OCL can be utilized not only to specify constraints, but for a number of different purposes (8, pp. 5-6):

- As a query language;
- To specify invariants on classes and types in the class model;
- To specify type invariant for stereotypes;
- To describe pre and post conditions on operations and methods;
- To describe guards;
- To specify target (sets) for messages and actions;
- To specify constraints on operations;
- To specify initial values or derivation rules for attributes or association ends for any expression over a UML model.

Any OCL expression indicates a value or an object in the system. For example, the expression “2+5” is a valid OCL expression, of type `Integer`, which represents the value “7”. When the value of an expression is of type `Boolean`, it can be utilized as an invariant (40).

Additionaly, only a subset of the OCL 2.0 language, named Essential OCL, can be used with EMOF (meta)models. This subset is based on the common core between UML 2.0 Infrastructure (12) and MOF 2.0 Core (4), because the full OCL specification can only be used with the UML language (8, pp. 1,171). The Essential OCL language is defined in 8, pp. 171-175. As we use the Ecore language (for Ecore, see section 2.2.5.1) to build the OntoUML metamodel, then all OCL expressions that we construct in this thesis are Essential OCL expressions.

2.2.3.1 OCL Examples

Now, we will get our examples of the OCL syntax from the tutorial 41. OCL has four basic primitive datatypes: `Boolean` (true, false), `Integer`, `Real` and `String`. Furthermore, OCL has the following comparators: <=, >= and =.

This language has the following operations for primitive types:

- `Integer`: *, +, -, /, `div()`, `abs()`, `mod()`, `max()`, `min()`, `sum()`, `sin()` and `cos()`.
- `Real`: *, +, -, /, `floor()`, `sum()`, `sin()` and `cos()`.
- `Boolean`: `and`, `or`, `xor`, `not`, `implies` and `if-then-else`.
- `String`: `concat()`, `size()`, `substring()`, `toInteger()` and `toReal()`.

In OCL, there is a class named `Collection` that is the abstract superclass of the classes `Set`, `OrderedSet`, `Bag` and `Sequence`. A `Set` is a collection without duplicates, having no order. `OrderedSet` is a collection without duplicates, having an order. The `Bag` class is a collection in which duplicates are allowed, having no order. Finally, a `Sequence` is a collection in which duplicates are allowed, having an order.

There are also operations for Collections:

- The number of elements in a collection: `size()`.
- The information of whether an object is part of a collection: `includes()`.
- The information of whether an object is not part of a collection: `excludes()`.
- The number of times that object occurs in a collection: `count()`.
- The information of whether all objects of a given collection are part of a specific collection: `includesAll()`.
- The information of whether none of the objects of a given collection are part of a specific collection: `excludesAll()`.
- The information if a collection is empty: `isEmpty()`.
- The information if a collection is not empty: `notEmpty()`.

The iterators over collections:

- The selection of a sub-collection: `select()`.
- When specifying a collection that is derived from some other collection, but which contains different objects from the original collection (*i.e.*, it is not a sub-collection) use: `collect ()`.
- The information of whether an expression is true for all objects of a given collection: `forAll()`.
- The addition of all elements of a collection (where the elements must be of a type supporting the + operation): `sum()`.

OCL collection operation examples:

- Specifying a sequence literal: `Sequence{1, 2, 3}`.
- Is a collection empty?: `Sequence{1, 2, 3}->isEmpty()`.
- Getting the size of a collection: `Sequence{1, 2, 3}->size()`.
- Please compare: "`Sequence{3, 3, 3}->size()`" returns `3` while "`Set{3, 3, 3}-> size()`" returns `1`.
- Nesting sequences: `Sequence{Sequence{2, 3}, Sequence{1, 2, 3}}`.
- Getting the first element of a sequence: `Sequence{1, 2, 3}->first()`.
- Getting the last element of a sequence: `Sequence{1, 2, 3}->last()`.
- Selecting all elements of a sequence that are smaller than 3: `Sequence{1, 2, 3, 4, 5, 6}->select( i | i <= 3)`.
- Rejecting all elements of a sequence that are smaller than 3: `Sequence{1, 2, 3, 4, 5, 6}->reject( i | i <= 3)`.
- Collect the names of all MOF classes: `MOF!Class.allInstances()->collect(e|e. name)`.
- Are all numbers in the sequence greater than 2?: `Sequence{12, 13, 12}->forAll( i | i>2 )`.
- Exists a number in the sequence that is greater than 2?: `Sequence{12, 13, 12}-> exists( i | i>2 )`.

- Concatenating Sequences: `Sequence{1, 2, 3}->union(Sequence{4, 5, 6})`.

OCL also enables one to formulate an if-clause. For example, if three is greater than two return "three is greater than two" else return "oh": `if 3 > 2 then 'three is greater than two'else 'oh'endif`

There are different operations to treat and analyze classes:

- The operation `oclIsTypeOf()` checks if a given instance is an instance of a certain type (and not of one of its subtypes or of other types).
- The operation `oclIsKindOf()` checks if a given instance is an instance of a certain type or of one of its subtypes.
- The operation `allInstances()` returns you all instances of a given Type.
- The operation `oclIsUndefined()` tests if the value of an expression is undefined (*e.g.*, if an attribute with the multiplicity zero to one is void or not. Please note: attributes with the multiplicity *n* are often represented with collections, which may be empty and not void).

Examples on OCL class operations:

- Please compare "`MOF!Attribute.oclIsKindOf(MOF!ModelElement)`" is true while "`MOF!Attribute.oclIsTypeOf(MOF!ModelElement)`" is false.

Finally, OCL comments start with two consecutive hyphens ("`--`") and end at the end of the line.

2.2.4 ATL

In the context of the MDE approach, model transformations are very important. ATL is a transformation language developed as a part of the ATLAS Model Management Architecture[8] (Bézivin *et al.*(1), 2004 apud Jouault *et al.*(2), 2006, p. 1) (AMMA) platform, which allows both imperative and declarative approaches to be used in transformation definitions (2).

The ATL language is based on the OCL specification for both its data types and its declarative expressions. But, there exist a few differences between the OCL specification and the current ATL implementation (42).

[8] http://atlanmod.emn.fr.

As shown in Fig. 2, an ATL transformation *mma2mmb.atl* from a source model *Ma* to a target model *Mb* is based on the metamodels *MMa* (in which *Ma* models are written), *MMb* (in which *Mb* models are written), and *MMATL* (the ATL metamodel). All metamodels must conform to the *MOF* metametamodel.

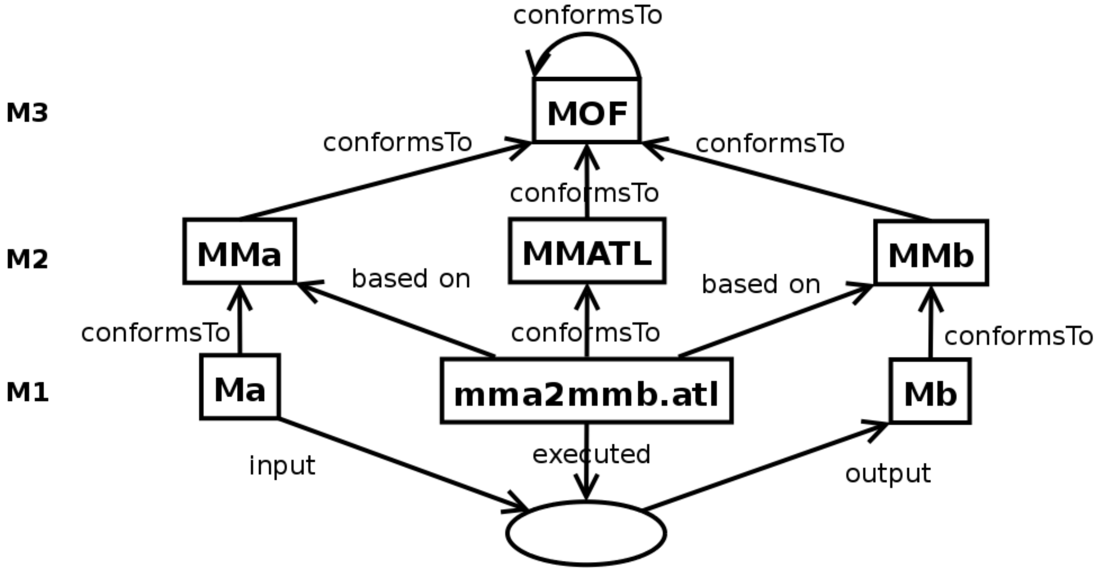


Figure 2: ATL overview (2, p. 719).

Also, the source and target models and metamodels may be expressed in XMI. Besides, one can also use the Kernel Meta Meta Model[9] (6) (KM3) notation (Jouault & Bézivin(43), 2006 apud Jouault *et al.*(2), 2006, p. 719) in order to write metamodels (2, p. 719).

One can find a detailed presentation of the language, examples and tutorials in Jouault & Kurtev (apud Jouault *et al.*(2), 2006, p. 719) and 45 and a formal specification of the ATL semantics in Ruscio *et al.* (apud Jouault *et al.*(2), 2006, p. 719) (2, p. 719).

This language is supported by a set of tools, named ATL Development Tools[10] (ADT), which is available as an Eclipse plug-in having an editor, a compiler and a debugger (2, p. 720).

2.2.5 Eclipse IDE

We use the Eclipse IDE mainly because it provides, in form of plug-ins, a set o functionalities that are usefull to this project, such as tools for creation and transformation of metamodels, and for the creation of graphical editors that are capable of syntactical verification.

In the words of The Eclipse Foundation (24):

[9] http://wiki.eclipse.org/KM3.

[10] http://www.eclipse.org/m2m/atl.

> "Eclipse is an open source community, whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle. The Eclipse Foundation is a not-for-profit, member supported corporation that hosts the Eclipse projects and helps cultivate both an open source community and an ecosystem of complementary products and services."(24)

In the Eclipse IDE, some functionalities are provided via independent frameworks, implemented as plug-ins. In this thesis, we use:

- Eclipse Modeling Framework[11] (3) (EMF);
- Graphical Modeling Framework[12] (GMF);
- Model Development Tools[13] (MDT);
- The ATLAS Transformation Language[14] (ATL) plug-in for Eclipse;
- The Alloy Analyzer plug-in for Eclipse[15];

2.2.5.1 EMF

In order to create the OntoUML metamodels, we have used the EMF framework. EMF is a framework for code generation that unifies three technologies: Java, eXtensible Markup Language[16] (XML) and UML, as shown in Fig. 3. It allows us to define a model by using one of these technologies (*e.g.*, as a Java interface, a XML Schema or a UML diagram) and then generate a corresponding model in any of the other technologies (37, p. 14).

The EMF models are written in a language named Ecore. Ecore is the name of the metamodel (implemented in Java) of the EMF core. There are small, mostly naming differences between Ecore and EMOF (a subset of the MOF 2.0 metamodel (4)). However, EMF can transparently read and write serializations of EMOF (47).

In the words of 37, p. 39:

> "MOF and Ecore have many similarities in their ability to specify classes and their structural and behavioral features, inheritance, packages, and reflection. They differ in the area of life cycle, data type structures, package relationships, and complex aspects of associations."(37, p. 39)

[11] http://www.eclipse.org/modeling/emf.
[12] http://www.eclipse.org/modeling/gmf.
[13] http://www.eclipse.org/modeling/mdt.
[14] http://www.eclipse.org/m2m/atl.
[15] http://alloy4eclipse.googlecode.com.
[16] http://www.w3.org/XML.

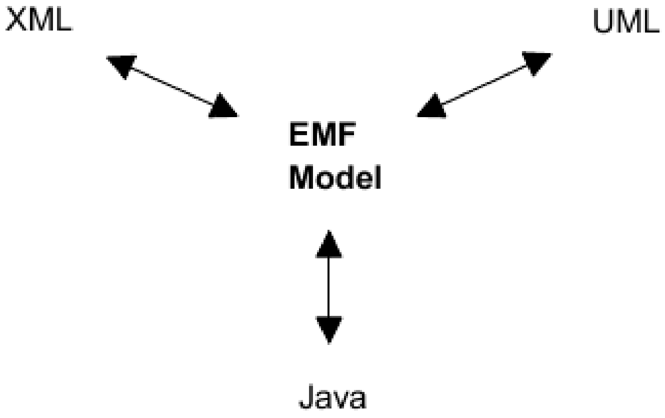


Figure 3: EMF unifies Java, XML, and UML (37, p. 14).

Now, we can update Fig. 3 in the Fig. 4.

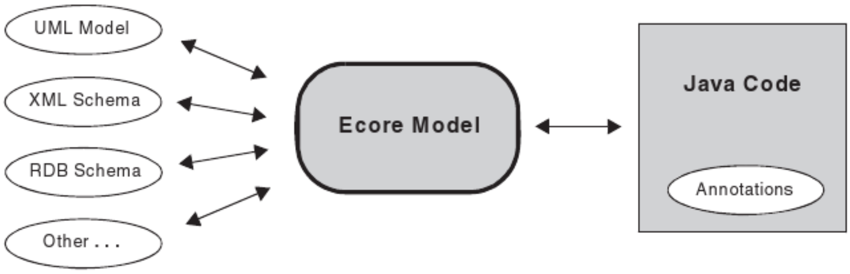


Figure 4: EMF framework (37, p. 23).

The EMF Eclipse plug-in also provide means for the creation and verification of Ecore (meta)models embedded with OCL expressions via the integration with the MDT framework, which provides an implementation of the OCL language for the assessment of queries, constraints and descriptions of operations within Ecore (meta)models.

Additionaly, as only Essential OCL can be used with EMOF (meta)models (see section 2.2.3), then all OCL expressions that we build in this thesis are Essential OCL expressions.

Fig. 5 shows an overview of the process of creating a graphical editor by using the EMF plug-in.

Firstly, we have to create an EMF project and build an Ecore metamodel for the chosen language. In this metamodel, we can also define some operations and some derived meta-attributes and meta-references (which, in Ecore, are called EOperations, EAttributes and EReferences, respectively) by including the OCL expressions.

Then, we have to automatically transform this metamodel into a Genmodel file. This file has properties that are responsible for customizing code generation for the Ecore file (3, p. 47). If the metamodel contains OCL expressions, then, as is described in 49, we have to set some variable

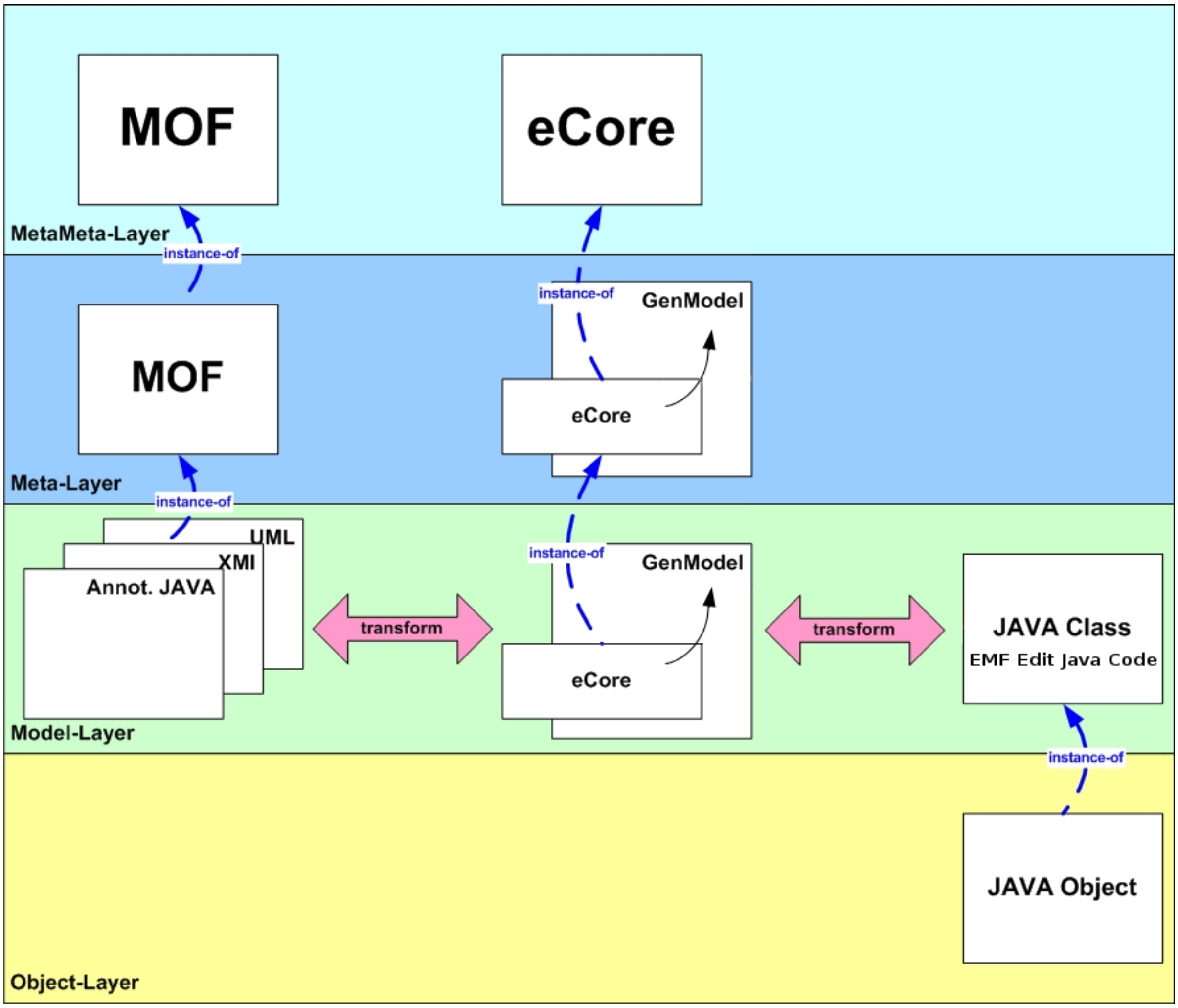


Figure 5: EMF overview (based on 48).

names in this Genmodel file, regarding the use of some Java Emitter Templates[17] (JET) files, which are needed in order to enable the EMF plug-in to handle, by means of the MDT plug-in, the OCL expressions that are in the Ecore metamodel.

Following 50, from the Genmodel, we perform two automatic transformations in order to get the Java code for the Ecore file and an EMF.Edit (51) framework, which provides generic reusable classes, which we use to build the graphical editor.

2.2.5.2 GMF

The GMF framework is utilized in the construction of graphical editors from Ecore metamodels. It serves as an agent between the EMF framework and the Graphical Editing Framework[18] (3) (GEF) framework, which is a plug-in that allows the construction of graphical editors from

[17] http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html.
[18] http://www.eclipse.org/gef.

Ecore metamodels, but of difficult utilization. Fig. 6 shows an overview of the process of creating a graphical editor by using the GMF plug-in.

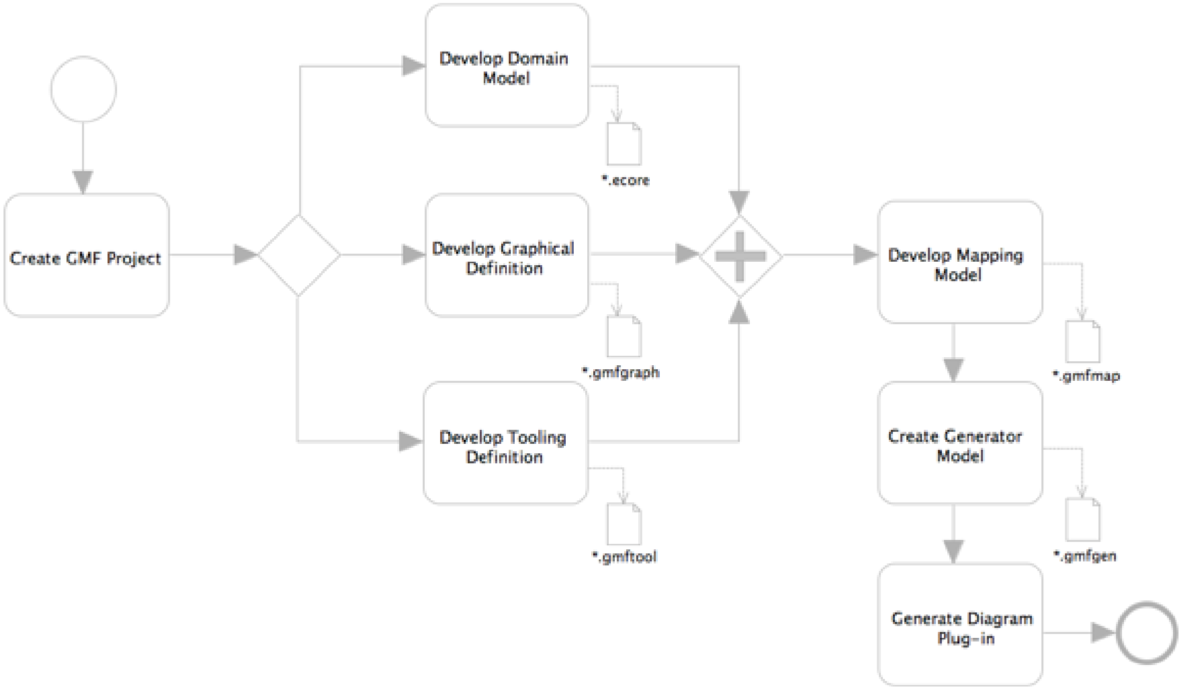


Figure 6: GMF overview (52).

Firstly, we create an EMF project and build an Ecore metamodel for the chosen language. After, we create a GMF project and two new files, which are based on the language metamodel: a GMFGraph file, which describes the visualization of the graphical elements of the editor; and a GMFTool file, which describes the tool palette that is utilized in order to instantiate the constructs of the language.

In order to create a mapping between the Ecore metamodel, the GMFGraph file and the GMFTool file, we create a GMFMap file. In this file, we map the elements from the Ecore metamodel to their visualization specified in the GMFGraph file and their creation tools specified in the GMFTool file. Also, in this file we can put some OCL constraints representing (i) the syntactical constraints of the language, which can be verified in live or in batch mode[19], and (ii) the automatic initialization or modification of some meta-attributes' values[20].

Once the creation of the GMFMap file is finished, we transform this file into a GMFGen file. This file is utilized in the automatic generation, by the GMF framework, of the Java code that implements the editor.

[19] Those verification modes are explained in subsections 4.6.1 and 4.6.3, respectively.

[20] See subsection 4.3.3.

2.3 The Logic-based Language Alloy

Alloy is a language for describing structural properties. It offers a set-based formula syntax by which one can express constraints that are suitable to a fully automatic semantic analysis. Its meaning is given by translation to a formally defined kernel (27, pp. 256-257). Alloy is based on the Z language (Spivey(53), 1989 apud Jackson(25), 2006, p. 1), selecting from Z those features that were considered essential for object modeling (27, p. 257).

The analysis of Alloy specifications is a form of constraint solving. This language supports two kinds of analysis: *simulation*, in which the consistency of an invariant or operation is demonstrated by generating an instance (a state or transition); and *checking*, in which a consequence of the specification is tested by attempting to generate a counterexample, *i.e.*, an structure that violates a given property of the specification. (27, p. 260)(25, p. 3)

There is a tool, named Alloy Analyzer (26), that is capable of simulating and checking Alloy specifications. If an Alloy specification has at least one instance, it is said to be consistent; when every possible assignment of values to the specification variables (respecting the variable type) is an instance, then the specification is valid. The negation of a valid specification is inconsistent. Therefore, in order to check an assertion, Alloy Analyzer looks for an instance of its negation; if one is found, it is a counterexample (27, p. 267).

This approach is sometimes called “lightweight formal methods”, because it tries to obtain the benefits of traditional formal methods, such as theorem proving techniques, at lower cost (25, p. XIII).

Regarding the syntax of Alloy, its specifications are basically composed of *signatures*, *fields*, *facts*, *predicates*, *functions* and *assertions*. In order to explain the basic Alloy constructs, we create two simple and equivalent Alloy specifications that are shown in Listings 1 and 2. In these Alloy specifications, we state that the classes Man and Woman partition the class Person, and that persons must have at least one parent and any number of children. But, for a specific person, his/her sets of parents and children must (i) not contain him/herself and (ii) have no intersection (which is the same as stating that the elements within these sets are pairwise disjoint).

In the Alloy language, a signature is a declaration of a set that can contain only urelements (which are called *atoms* (27, pp. 35-36) in Alloy). Some examples are the signatures Person, Man and Woman, shown in lines 1 and 7 of Listing 1, and 1 and 7 of Listing 2.

Moreover, Alloy allows the definition of subsignatures (subsets) by the keywords “`in`”, which collapses the $\in$ and $\subseteq$ set-theoretic operators (27, p. 55) (see line 7 of Listing 2), and "`extends`", which is used to declare pairwise disjoint subsignatures of a signature (27, p. 91) (see line 7 of Listing 1).

The keyword "`abstract`" (line 1 of Listing 1) indicates that when an "`abstract`" signature S is extended (by using "`extends`") by other subsignatures $S_1, \ldots, S_n$, then all the atoms of S must be atoms of at least one of the $S_1, \ldots, S_n$ signatures (27, p. 91), *i.e.*, when S is abstract, the subsignatures that extends S also partition S.

Furthermore, all top-level signatures (*i.e.*, signatures that are subsignatures of no signature (27, p. 91)) are pairwise disjoint (27, p. 91). In Alloy, relations are sets of tuples, which may be of any finite arity (27, p. 43), but containing only atoms (also called flat relations) (27, p. 41). They must be declared as fields within signatures (27, p. 95). Some examples are the relations parent and children, shown in lines 2 and 3 of Listing 1, and 2 and 3 of Listing 2.

One can also use the "`in`" keyword to model subrelations, and the "`disj`" keyword to state that some relations are pairwise disjoint (27, p. 98) (see lines 5 of Listing 1, and 5 and 8 of Listing 2).

Besides, Alloy facts are logical statements about signatures and relations that are always true for the whole specification (27, p. 117) (see lines 9 of Listing 1, and 8 and 10 of Listing 2). When created within signatures, they are called *signature facts* and are implicitly universally quantified over all the atoms of the signature (27, pp. 18,118,269). Some examples are shown in lines 5 of Listing 1, and 5 of Listing 2.

Predicates are reusable parametrized constraints that only hold when invoked (27, pp. 121,123). Some example are shown in lines 8 and 9 of Listing 1, and 9 and 10 of Listing 2. Functions are reusable Alloy expressions (27, p. 121). Examples of functions that return the set of grandparents of a specific person are shown in lines 10 and 11 of Listing 1, and 11 and 12 of Listing 2. Finally, assertions are constraints that are intended to follow from the specification and that are meant to be *checked* against counterexamples (27, p. 124). Examples asserting that no person is one of his/her own grandparents are shown in lines 11 of Listing 1, and 12 of Listing 2. This assertion do not follow from these simple Alloy specifications, so Alloy Analyzer is able to find counterexamples in which a person is one of his/her own grandparents.

Listing 1: An Alloy specification.

```
1 abstract sig Person {
2   parents: some Person,
3   children: set Person
4 }{
  disj[parents,children]
}
sig Man, Woman extends Person {}
pred irreflexive [r: univ -> univ] {no iden & r}
fact {irreflexive[parents] and irreflexive[children]}
fun grandparents [x: Person] : set (Person) {(x.parents).
    parents}
check {no x: Person | x in grandparents[x]}
```

Listing 2: An Alloy specification.

```
sig Person {
  parents: some Person,
  children: set Person
}{
  disj[parents,children]
}
sig Man, Woman in Person {}
fact {disj[Man,Woman]}
pred irreflexive [r: univ -> univ] {no iden & r}
fact {irreflexive[parents] and irreflexive[children]}
fun grandparents [x: Person] : set (Person) {(x.parents).
    parents}
check {no x: Person | x in grandparents[x]}
```

It is worth to notice that the Alloy Analyzer can only handle *formulæ* that involve higher-order quantifications if they can be eliminated by a scheme known as "skolemization", which turns a quantified variable into a free variable whose value can then by found by constraint solving. Therefore, in general, Alloy Analyzer deals with first-order specifications (25, p. 72-73).

The search for instances is conducted in a space whose dimensions are specified by the user in a "scope", which assigns a bound to the number of objects of each type (25, p. 3). Therefore, Alloy's analysis is limited to a finite scope that bounds the sizes of the carrier sets of the basic types. An instance is within a scope of *k* if it assigns to each type a set consisting of no more than *k atoms*. If the analysis succeeds in finding an instance to an specification, consistency is demonstrated. Failure to find an instance within a given scope, however, does not prove that the specification is inconsistent, because, since the kernel in which Alloy is based is undecidable, it is impossible to determine automatically whether an Alloy specification is valid or consistent

(27, p. 267)(25, p. 259). In other words, the inexistence of an instance that fits in a scope *k* does not imply that there is no scope larger than *k* in which an instance exists. In the words of Jackson in 27, p. 274:

> “Ideally, the language would be decidable. Unfortunately, the most elementary calculus that involves relations is undecidable - even Tarski’s relational calculus, which has no quantifiers and is strictly less expressive than first order logic. Some compromise is thus inevitable. Alloy’s analysis finds models of formulas: that is, assignments of values to variables for which the formula is true. When the formula is the negation of a theorem, its models are counterexamples; when the formula is a state invariant or operation, the models are samples (either instances of the state or transitions). The analysis is guaranteed to be sound, in the sense that a model returned will indeed be a model. There are therefore no false alarms, and samples are always legitimate (and demonstrate consistency of the invariant or operation). On the other hand, when the analysis does not return a model, one cannot conclude that none exists. The validity of a theorem cannot be guaranteed, and an invariant or operation that appears to be inconsistent may in fact be consistent.” (27, p. 274)
> “The analysis works by considering all potential models up to a given size, specified by a scope that limits the number of atoms in each primitive type. In practice, it seems that most interesting properties, whether samples or counterexamples, can be illustrated within a small scope. So the absence of a model within a scope gives some empirical evidence that none exists, becoming more credible as the scope is increased.” (27, p. 274)
> “The analysis is explained elsewhere (54). In short, the values of a relation are viewed as adjacency matrices. Each relational variable is encoded as a matrix of boolean variables whose dimensions are determined by the scope, and a boolean formula is constructed whose models correspond to models of the original formula. An off-the-shelf Boolean satisfiability problem (SAT) solver is used to find solutions to the boolean formula.” (27, p. 274)

Therefore, the Alloy Analyzer translates constraints to be solved from Alloy specifications into boolean constraints, which are fed to a SAT solver in order to verify boolean satisfiability (25, p. XII). Furthermore, by constraining the search to a finite scope, the analysis of Alloy specifications is decidable, and as a SAT problem, it is NP-complete. From version four, the Alloy Analyzer translates constraints to be solved from Alloy into boolean constraints, which are fed to the SAT-based model finder Kodkod[21]. From (25, p. XII):

> “As solvers get faster, so Alloy’s analysis gets faster and scales to larger problems. Using the best solvers of today, the analyzer can examine spaces that are several hundred bits wide (that is, of 10^{60} cases or more).” (25, p. XII)

Moreover, when translating Alloy specifications into boolean *formulæ*, Alloy Analyzer applies a variety of optimizations, where the most important is *symmetry breaking*. Every Alloy

[21] http://alloy.mit.edu/kodkod.

specification has an intrinsic symmetry given by the possibility to permute the atoms in any instance of a command, without ceasing to satisfy the Alloy specification. So, the space of assignments (possible solutions) can be divided into equivalence classes, and the solver has to search for only one assignment at each equivalence class (25, p. 151).

In pace with Daniel Jackson (27, p. 260), we believe that “simulation helps catch errors of overconstraint, by reporting, contrary to the user’s intent, that no instance exists within the finite bounds of a given “scope””, or errors of underconstraint, “by showing instances that are acceptable to the specification but which violate an intended property.”.

Finally, the purpose of the Alloy language is well summarized by Jackson in (27, p. 260):

> “Together, the two analysis enable an incremental process of specification. One starts with a minimal model, and performs a variety of simulations to detect overconstraint. Intended consequences are formulated, with counterexamples suggesting additional constraints to be added to the specification. This process helps produce a specification that has the desired properties and no more.” (27, p. 260)

3 Unified Foundational Ontology and OntoUML Language

Guizzardi, in 11, defines UFO, contributing to the definition of general ontological foundations for the area of conceptual modeling. UFO is intended to be 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" (11). In 11, the author also does exactly that by proposing an ontologically well-founded version of the class diagram part of UML 2.0, dubbed *OntoUML*. The ontological categories comprising UFO are motivated by a number of theories in formal ontology, philosophical logics, cognitive science and linguistics. Moreover, the distinctions between these categories are motivated by a number of formal meta-properties, some of which will be discussed in the sequel.

The UML language has some well know problems regarding the representation of part-whole relationships, as it collapses many types of parthood relations into shareable or composite associations[1] (11, pp. 341-352), and the fact that it is not capable of handling any modal characterization, allowing one to make mistakes, such as the use of subtyping to represent alternative allowed types (20, p. 123) (see subsection 4.6.1) and the creation of wholes that have only one part, disobeying the weak supplementation principle (*i.e.*, if x is a part of y then there must be a z disjoint of x, which is also a part of y, see subsection 4.6.3).

In order to formally assess the quality of UML 2.0, 11, pp. 28-36 proposes a framework for assessment of modeling languages. This framework proposes that in order to assess the quality of a language, we shall create mappings between the metamodel of the language and the metamodel of a suitable foundational ontology.

Briefly, a language will be considered suitable when all of its concepts have an unique counterpart in the foundational ontology. If there are disjoint concepts in the ontology that are represented by the same language construct, there will be the case of *non-lucidity*, also called *construct overload*, which leads to ambiguity in the produced models. If there are constructs in

[1] As explained in section 3.3, OntoUML consider four sorts of conceptual parthood relations, *viz.* subQuantityOf, subCollectionOf, memberOf and componentOf, regarding the types of their relata.

the language that represent no concept in the ontology, there will be the case of *unsoundness*, also called *construct excess*, which leads to uncertain in the produced models. If there are concepts in the ontology that are represented by more than one language construct, there will be the case of *non-laconicity*, also called *construct redundancy*, which leads to unnecessary complexity in the language. Finally, if there are concepts in the ontology that are not represented by some language construct, there will be the case of *incompleteness*, which leads to incompleteness in the produced models (11).

In the following, we show the results obtained in 11 by applying this framework in the assessment of the part of the UML 2.0 regarding class diagrams and using UFO (proposed from chapters 4 to 7 of 11, pp. 95-309) as the reference foundational ontology. The purpose of this assessment and reconstruction of UML 2.0 was to obtain an ontologically well-founded language (later termed OntoUML) for building conceptual models, and, in particular, domain ontologies.

Therefore, in this chapter, we briefly present the UFO theory and some excerpts of the OntoUML metamodel, so one can understand key concepts of OntoUML that we will make use from now on. We will not present the assessment of UML 2.0 by the application of the framework. One can get more details in 11, pp. 311-352.

In the assessment of UML 2.0, it was used only a fragment of the UFO ontology regarding endurants (also named continuants, *e.g.*, objects), as opposed to perdurants (also named occurents, *e.g.*, events and processes) (11, p. 211). As is done in chapter 8 of 11, pp. 311-352, in the following three sections we present the UFO and OntoUML metamodels divided in three fragments, *viz.* Classes and Generalization (section 3.1), Classifiers and Properties (section 3.2) and Aggregation and Composition (section 3.3).

Since OntoUML is a modeling language whose metamodel is designed to be isomorphic to the UFO ontology, the leaf ontological distinctions in the metamodel of UFO appear as modeling primitives in the language.

3.1 Classes and Generalization

The Fig. 7 represents the UFO's metamodel excerpt regarding Classes and Generalization. In this excerpt, Universals are space-time independent pattern of features, which can be realized in a number of different individuals (instances). Universals can be Monadic Universals (*e.g.*, Person, Sand, Forest, Woman, Teenager, Student, RelationalEntity, Supplier, Buoyancy and ChairColor) or Relations.

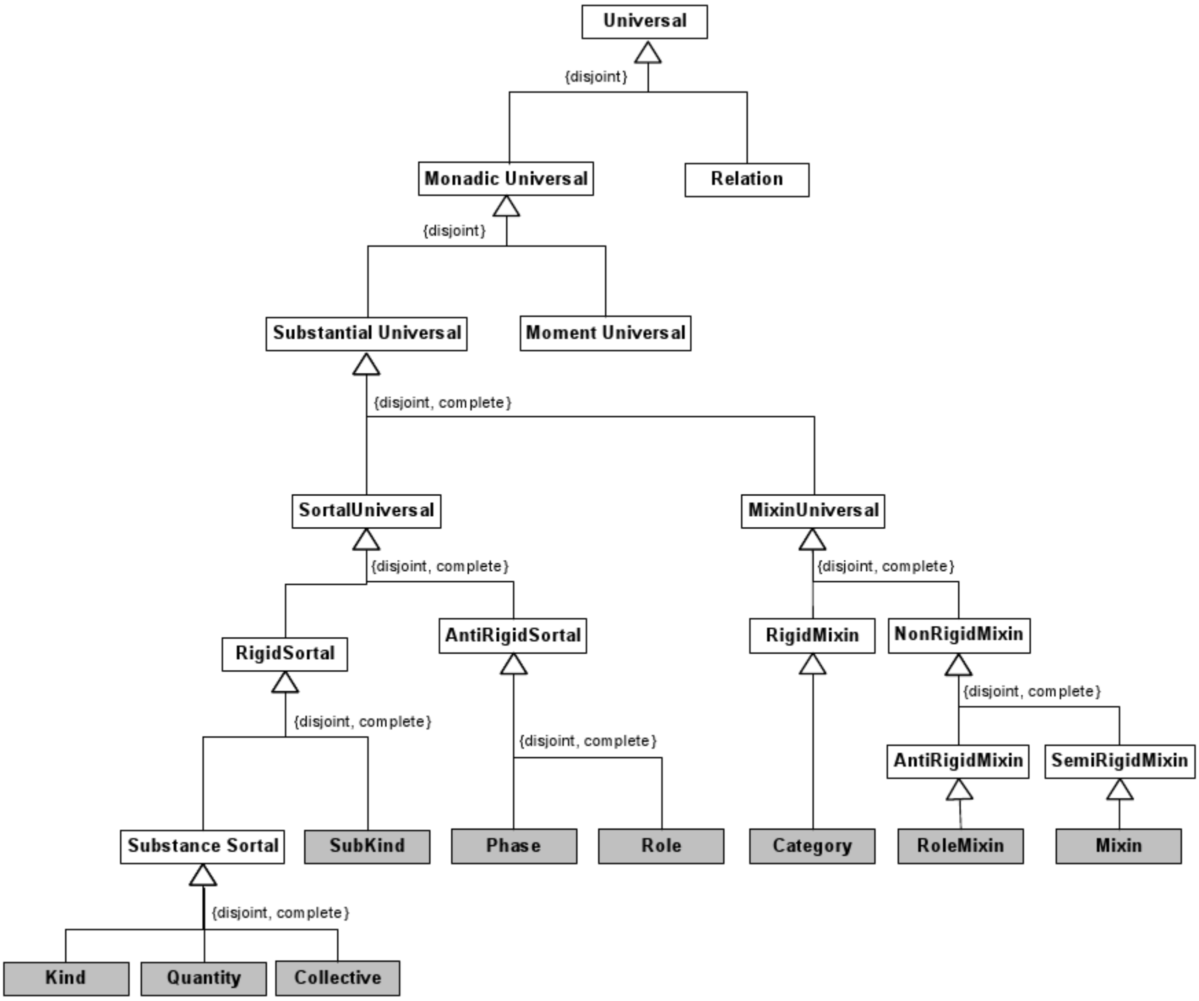


Figure 7: UFO excerpt regarding Substantial Universals (11, p. 315).

Monadic Universals can be Substantial Universals or Moment Universals. The distinction between Substances and Moments is based on the formal notion of existential dependence, a modal notion that can be briefly defined as follows:

> **Definition 1 (Existential Dependence)**: Let the predicate ε denote existence[2]. We have that an individual x is existentially dependent on another individual y iff, as a matter of necessity, y must exist whenever x exists, or formally: $ed(x,y) \triangleq \Box(\varepsilon(x) \rightarrow \varepsilon(y))$. ■ (55, p. 11)

Substances are existentially independent individuals, *i.e.*, there is no entity y *disjoint* from x that must exist whenever a Substance x exists. The disjointness constraint is necessary to exclude the trivial examples, such as an instance in which an individual is existentially dependent on its essential parts (see discussion latter in this section). Let $\leq$ represent the (improper) part of relation. This constraint can be formalized as follows: $disjoint(x,y) \triangleq \neg\exists z((z \leq x) \land (z \leq y))$ and $\forall x,y((Substance(x) \land Substance(y) \land disjoint(x,y)) \rightarrow (\neg ed(x,y) \land \neg ed(y,x)))$.

[2] In an actualist system, the existence operator ε can be explicitly defined such that $\varepsilon(x) \triangleq \exists y(y = x)$.

Also, from 11, p. 95, “Substantials are entities that persist in time while maintaining their identity”. Examples of Substances include ordinary mesoscopic objects such as an individual person, a house, a hammer, a car, but also the so-called *Fiat Objects* such as the North-Sea and its proper-parts, postal districts and a non-smoking area of a restaurant.

Therefore, Substantial Universals are Universals whose instances are Substances, *i.e.*, whose instances are existentially independent individuals that persist in time while maintaining its identity (*e.g.*, Person, Sand, Forest, Woman, Teenager, Student, RelationalEntity, Supplier and Buoyancy).

Conversely, a Moment is an individual that can only exist in other individuals, in which it *inheres* (11, p. 213) (see section 3.2 for more details) and to which it is existentially dependent. Therefore, Moment Universals are Universals whose instances are Moments (*e.g.*, ChairColor). A moment can inhere on one single individual (*e.g.*, the color of a chair, an electric charge) or on multiple individuals (*e.g.*, a covalent bond, a purchase order, a marriage), in which case they are named Relational Moments or simply Relators. The particular sort of existential dependence relation connecting a relator to the individuals it is dependent on is the formal relation of *mediation* (*m*), which is defined in section 3.2.

Substantial Universals can be Sortal Universals or Mixin Universals. Sortal Universals are Universals that provide a principle of individuation and identity to its instances (its particulars)(11, p. 98) (*e.g.*, Person, Sand, Forest, Woman, Teenager, Student, RelationalEntity, Supplier and Buoyancy). Sortal Universal can be specialized in Rigid Sortal (*e.g.*, Person, Sand, Forest and Woman) and AntiRigid Sortal (*e.g.*, Teenager and Student), for rigidity and anti-rigidity, see Definitions 2 and 3 respectively.

> **Definition 2 (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': $R(\mathrm{T}) \triangleq \Box(\forall x(\mathrm{T}(x) \rightarrow \Box(\mathrm{T}(x))))$. ■ (55, p. 9)
> **Definition 3 (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: $AR(\mathrm{T}) \triangleq \Box(\forall x(\mathrm{T}(x) \rightarrow \Diamond(\neg \mathrm{T}(x))))$. ■ (55, p. 9)

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 (11, p. 100) (*e.g.*, Person, Sand and Forest). A SubKind is a rigid sortal that inherits its identity principle from a Kind that is one of its supertypes (for the definition of subtyping, see Definition 4) (11, p. 108) (*e.g.*, Woman, which inherits its identity principle from Person).

Regarding generalizations, both 12, p. 64 and 13, p. 72 states that:

> “Where a generalization relates a specific classifier to a general classifier, each instance of the specific classifier is also an instance of the general classifier. Therefore, features specified for instances of the general classifier are implicitly specified for instances of the specific classifier. Any constraint applying to instances of the general classifier also applies to instances of the specific classifier.” (12, p. 64),(13, p. 72)

Therefore, based on 56, we define generalization as: **Definition 4 (subtype)**: Where a generalization relates a specific non relational Classifier C_1 to a general non relational Classifier C_2, each instance x of C_1 is also an instance of C_2 in every world w in which x is an instance of C_1: $subtype(C_1, C_2) \triangleq \Box(\forall x(C_1(x) \rightarrow C_2(x)))$. ■

Regarding the GeneralizationSet meta-attributes *isCovering* and *isDisjoint*, 13, p. 75 defines them in the following way:

- “isCovering : Boolean
  Indicates (via the associated Generalizations) whether or not the set of specific Classifiers are covering for a particular general classifier. When isCovering is true, every instance of a particular general Classifier is also an instance of at least one of its specific Classifiers for the GeneralizationSet. When isCovering is false, there are one or more instances of the particular general Classifier that are not instances of at least one of its specific Classifiers defined for the GeneralizationSet. For example, Person could have two Generalization relationships each with a different specific Classifier: Male Person and Female Person. This GeneralizationSet would be covering because every instance of Person would be an instance of Male Person or Female Person. In contrast, Person could have a three Generalization relationship involving three specific Classifiers: North American Person, Asian Person, and European Person. This GeneralizationSet would not be covering because there are instances of Person for which these three specific Classifiers do not apply. The first example, then, could be read: any Person would be specialized as either being a Male Person or a Female Person - and *nothing else*; the second could be read: any Person would be specialized as being North American Person, Asian Person, European Person, or something else. Default value is *false*.”. (13, p. 75)
- “isDisjoint : Boolean
  Indicates whether or not the set of specific Classifiers in a Generalization relationship have instance in common. If isDisjoint is true, the specific Classifiers for a particular GeneralizationSet have no members in common; that is, their intersection is empty. If isDisjoint is false, the specific Classifiers in a particular GeneralizationSet have one or more members in common; that is, their intersection is *not* empty. For example, Person could have two Generalization relationships, each with the different specific Classifier: Manager or Staff. This would be disjoint because every instance of Person must either be a Manager or Staff. In contrast, Person could have two Generalization relationships involving two specific (and non-covering) Classifiers: Sales Person and Manager. This GeneralizationSet would not be disjoint because there are instances

> of Person that can be a Sales Person *and* a Manager. Default value is *false*." (13, p. 75)

Also, when a GeneralizationSet has the value "true" in both *isCovering* and *isDisjoint* meta-attributes, then it is know as a "partition" (13, p. 78).

Therefore, we define covering, disjoint and partition GeneralizationSets that contains generalizations between non relational Classifiers, such as Classes or Datatypes, as follows:

- **Definition 5 (covering)**:
  For a GeneralizationSet GS, we have a non relational Classifier (such as a Class or a Datatype) C_{ST} that is the common supertype that is referred by all Generalizations referred by GS, and also some non relational Classifiers $\mathrm{C}_1,\ldots,\mathrm{C}_n$ $(n \in \mathbb{N}^*)$ that are the subtypes of C_{ST} that are referred by the Generalizations referred by GS. When the value of the meta-attribute *isCovering* of this GeneralizationSet GS is "true", then every instance of the general non relational Classifier C_{ST} is also an instance of at least one non relational Classifier in $\{\mathrm{C}_1,\ldots,\mathrm{C}_n\}$: $CoveringGeneralizationSet(\mathrm{C}_{ST},\mathrm{C}_1,\ldots,\mathrm{C}_n) \triangleq ((\bigwedge_{1\leq i\leq n} (\mathrm{C}_i(x) \rightarrow \mathrm{C}_{ST}(x))) \wedge (\mathrm{C}_{ST}(x) \rightarrow (\bigvee_{1\leq j\leq n} \mathrm{C}_j(x))))$. ■

- **Definition 6 (disjoint)**:
  For a GeneralizationSet GS, we have a non relational Classifier (such as a Class or a Datatype) C_{ST} that is the common supertype that is referred by all Generalizations referred by GS, and also some non relational Classifiers $\mathrm{C}_1,\ldots,\mathrm{C}_n$ $(n \in \mathbb{N}^*)$ that are the subtypes of C_{ST} that are referred by the Generalizations referred by GS. When the value of the meta-attribute *isDisjoint* of this GeneralizationSet GS is "true", then every instance of a non relational Classifier C_i $(i \in \{1,\ldots,n\})$ is pairwise disjoint with any instance of any non relational Classifier C_j $(j \in (\{1,\ldots,n\} - \{i\}))$: $DisjointGeneralizationSet(\mathrm{C}_{ST},\mathrm{C}_1,\ldots,\mathrm{C}_n) \triangleq (\bigwedge_{1\leq i\leq n} (\mathrm{C}_i(x) \rightarrow \mathrm{C}_{ST}(x) \wedge (\bigwedge_{1\leq j\leq n \mid j\neq i} \neg \mathrm{C}_j(x))))$. ■

- **Definition 7 (partition)**:
  For a GeneralizationSet GS, we have a non relational Classifier (such as a Class or a Datatype) C_{ST} that is the common supertype that is referred by all Generalizations referred by GS, and also some non relational Classifiers $\mathrm{C}_1,\ldots,\mathrm{C}_n$ $(n \in \mathbb{N}^*)$ that are the subtypes of C_{ST} that are referred by the Generalizations referred by GS. When both values of the meta-attributes *isCovering* and *isDisjoint* of this GeneralizationSet GS are "true", then every instance of the general non relational Classifier C_{ST} is also an instance of exactly one non relational Classifier in $\{\mathrm{C}_1,\ldots,\mathrm{C}_n\}$: $PartitionGeneralizationSet(\mathrm{C}_{ST},\mathrm{C}_1,\ldots,\mathrm{C}_n) \triangleq$

$((\bigwedge_{1\leq i\leq n} \mathrm{C}_i(x) \rightarrow \mathrm{C}_{ST}(x)) \wedge (\mathrm{C}_{ST}(x) \rightarrow (\bigoplus_{1\leq j\leq n} \mathrm{C}_j(x))))$, where $\oplus$ represents the exclusive disjunction. ■

Returning to the UFO metamodel, Substance Sortal can be specialized into Kind, Quantity and Collective. Kinds are Substance Sortals that provides identity principles for its instances (11, p. 108) (*e.g.*, Person). Quantities are maximally self-connected objects (11, pp. 179-180) that have no *individuation* and *counting principles* (11, p. 174) (*e.g.*, Sand). Collectives are collections of entities connected by an unifying relationship (11, p. 181) (*e.g.* Forest).

Within the AntiRigidSortal category, we have a further distinction between Phases and Roles. Both Phases and Roles are specializations of rigid universals (Kinds/SubKinds). However, they are differentiated w.r.t. their specialization conditions. For the case of Phases, the specialization condition is always an intrinsic one, in other words, a Phase is a type an object instantiates in a period of time due to an intrinsic characteristic (11, p. 104) (*e.g.*, in Fig. 27, a Child is a LivingPerson whose age is within a certain range; likewise, a LivingPerson is a Person who has the property of being alive). Contrariwise, the specialization condition of a Role is a relational one, *i.e.*, a Role is a type an entity instantiates in a certain context (when mediated by a Relator (11, p. 294)[3]), as the participation in an event or relationship (*e.g.*, in Fig. 27, the Role Student, played by a LivingPerson who is enrolled in (has a *study* relation to) a School). Formally speaking, this distinction is based on a meta-property named Relational Dependence, which is defined in Definition 8:

> **Definition 8 (Relational Dependence)**: A type T is relationally dependent on another type P via relation R iff for every instace x of T, there is an instance y of P such that x and y are related via R: $R(\mathrm{T},\mathrm{P},\mathrm{R}) \triangleq \Box(\forall x(\mathrm{T}(x) \rightarrow \exists y(\mathrm{P}(y) \wedge \mathrm{R}(x,y))))$. ■ (55, p. 10)

Also, as discussed in 11, p. 103, *formulæ* 9, 10, Phases are always defined in a partition set:

> "**Definition 4.1 (Extension functions):** Let $\mathcal{W}$ be a non-empty set of possible worlds and let $w \in \mathcal{W}$ be a specific world. The extension function $ext_w(\mathrm{G})$ maps a universal G to the set of its instances in world w. The extension function ext(G) provides a mapping to the set of instances of the universal G that exist in all possible worlds, such that
> 1. $\mathrm{ext}(\mathrm{G}) = \bigcup_{w\in\mathcal{W}} \mathrm{ext}_w(\mathrm{G})$" (11, pp. 100-101, Definition 4.1)
> "... Let ‹$\mathrm{P}_1 \ldots \mathrm{P}_\mathrm{n}$› be a phase-partition that restricts the sortal S. Then we have that for all $w \in \mathcal{W}$:
> 9. $\mathrm{ext}_w(\mathrm{S}) = \bigcup_{\mathrm{P_i}\in\langle p_1 \ldots p_n\rangle} \mathrm{ext}_w(\mathrm{P_i})$
> and for all $\mathrm{P_i}, \mathrm{P_j} \in$‹$\mathrm{P}_1 \ldots \mathrm{P}_\mathrm{n}$› (with $\mathrm{i} \neq \mathrm{j}$) we have that
> 10. $\mathrm{ext}_w(\mathrm{P_i}) \cap \mathrm{ext}_w(\mathrm{P_j}) = \emptyset$" (11, p. 103)

[3] For an explanation on the mediation relation and the metaclass Relator, see section 3.2.

Because, from formula 9 in this citation, the GeneralizationSet containing the Phases is covering, and from formula 10, it is disjoint.

For instance, in Fig. 27, the universals Child, Teenager and Adult define a phase partition for the Phase LivingPerson. As consequence, we have that in an each world *w*, every LivingPerson is either a Child, a Teenager or an Adult in *w* and never more than one of these.

Additionally, from 11, p. 104 we have that:

> “Finally, it is always possible (in the modal sense) for an instance x of S to become an instance of each $\mathrm{P_i}$, i.e., for any $\mathrm{P_i} \in \langle \mathrm{P_1} \ldots \mathrm{P_n} \rangle$ which restricts S, and for any instance x such that $x \in \mathrm{ext}_w(\mathrm{S})$, there is a world $w' \in \mathcal{W}$ such that $x \in \mathrm{ext}_{w'}(\mathrm{P_i})$. This is equivalent of stating that for any $\mathrm{P_i} \in \langle \mathrm{P_1} \ldots \mathrm{P_n} \rangle$ the following holds
> 11. ext(S) = ext($\mathrm{P_i}$)” (11, p. 104)

These Phase constraints are formalized by Definition 9: **Definition 9 (Phase Partition)**: If in a world *w*, x is an instance of a SortalClass SC that is partitioned in Phase subclasses $\langle \mathrm{P}_1,\ldots,\mathrm{P}_n \rangle$, then, for every P_i such that $i \in \{1,\ldots,n\}$ there must exist at least one world w' in which x instantiates P_i: $\mathit{PhasePartition}(\mathrm{SC},\mathrm{P}_1,\ldots,\mathrm{P}_n) \triangleq ((\bigwedge_{1\leq i\leq n} (\mathrm{P}_i(x) \rightarrow \mathrm{SC}(x) \wedge (\bigwedge_{1\leq j\leq n \mid j\neq i} \Diamond \mathrm{P}_j(x)))) \wedge (\mathrm{SC}(x) \rightarrow (\bigoplus_{1\leq k\leq n} \mathrm{P}_k(x))))$. ■

In summary, in the model depicted in Fig. 27, the following example highlights the modal distinction between the rigid universal (Kind) Person, the (Role) universal Student, and the (Phase) universal Teenager. Suppose they are all instantiated by the individual John in a given circumstance. Whilst John can cease to be a Student and a Teenager (and there were circumstances in which John was none of the two), he cannot cease to be a Person. In other words, in a conceptualization that models Person as a Kind and Student as a Role, while the instantiation of the role Student has no impact on the identity of an individual, if an individual ceases to instantiate the Kind Person, then it ceases to exist as the same individual. Moreover, John instantiates the Phase Teenager and can cease to instantiate it due to changes of its age (an intrinsic property). John also contingently instantiates Student and can cease to instantiate it, but now motivated to a change in a relational property. Furthermore, 20, p. 117 formally proves that a rigid universal cannot have as its superclass an anti-rigid one. Consequently, a Role cannot subsume a Kind in UFO.

Mixin Universals are abstractions of properties that are common to multiple disjoint types (11, p. 112) (*e.g.*, RelationalEntity, Supplier and Buoyancy). They can be Rigid Mixins (*e.g.*, RelationalEntity) or NonRigid Mixins (*e.g.*, Supplier and Buoyancy). Rigid Mixins can be specialized into Categories, which classify entities that instantiate different Kinds, but share some essential characteristic (11, p. 112) (*e.g.*, RelationalEntity as a generalization of Person and Intelligent Agent). NonRigid Mixins represent characteristics that are essential to some of its instances and accidental to others.

NonRigid Mixins can be classified as AntiRigid Mixins (*e.g.*, Supplier) or SemiRigid Mixins (*e.g.*, Buoyancy). AntiRigid Mixins can be specialized into Role Mixins, which represents abstractions of common properties of Roles (11, p. 112) (*e.g.*, Supplier). SemiRigid Mixins can be specialized into Mixins, which represent non-rigid non-sortal entities, representing properties that are essential to some of its instances and accidental to others (11, p. 113) (*e.g.*, Buoyancy, which is an essential characteristic for a boat but an accidental one for a chair).

By applying the framework for assessment of modeling languages proposed in 11, pp. 28-36 and using the excerpt of UFO shown in Fig. 7, 11, p. 314 revise the excerpt of the UML 2.0 metamodel shown in Fig. 8, creating the excerpt of the OntoUML metamodel pictured by Fig. 9.

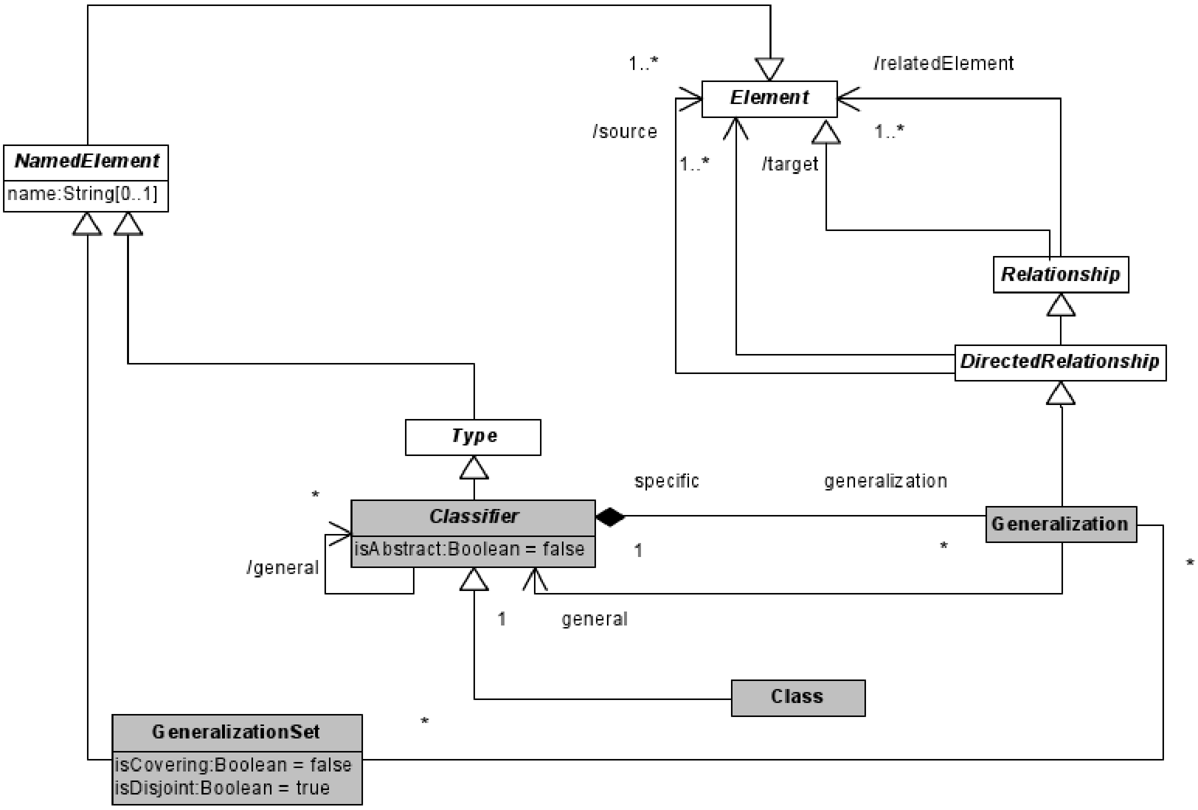


Figure 8: Excerpt from the UML metamodel featuring the metaclasses Classifier, Class, Generalization and GeneralizationSet (11, p. 312).

OntoUML profile regarding the categories depicted in Fig. 9

Metaclass: Substance Sortal

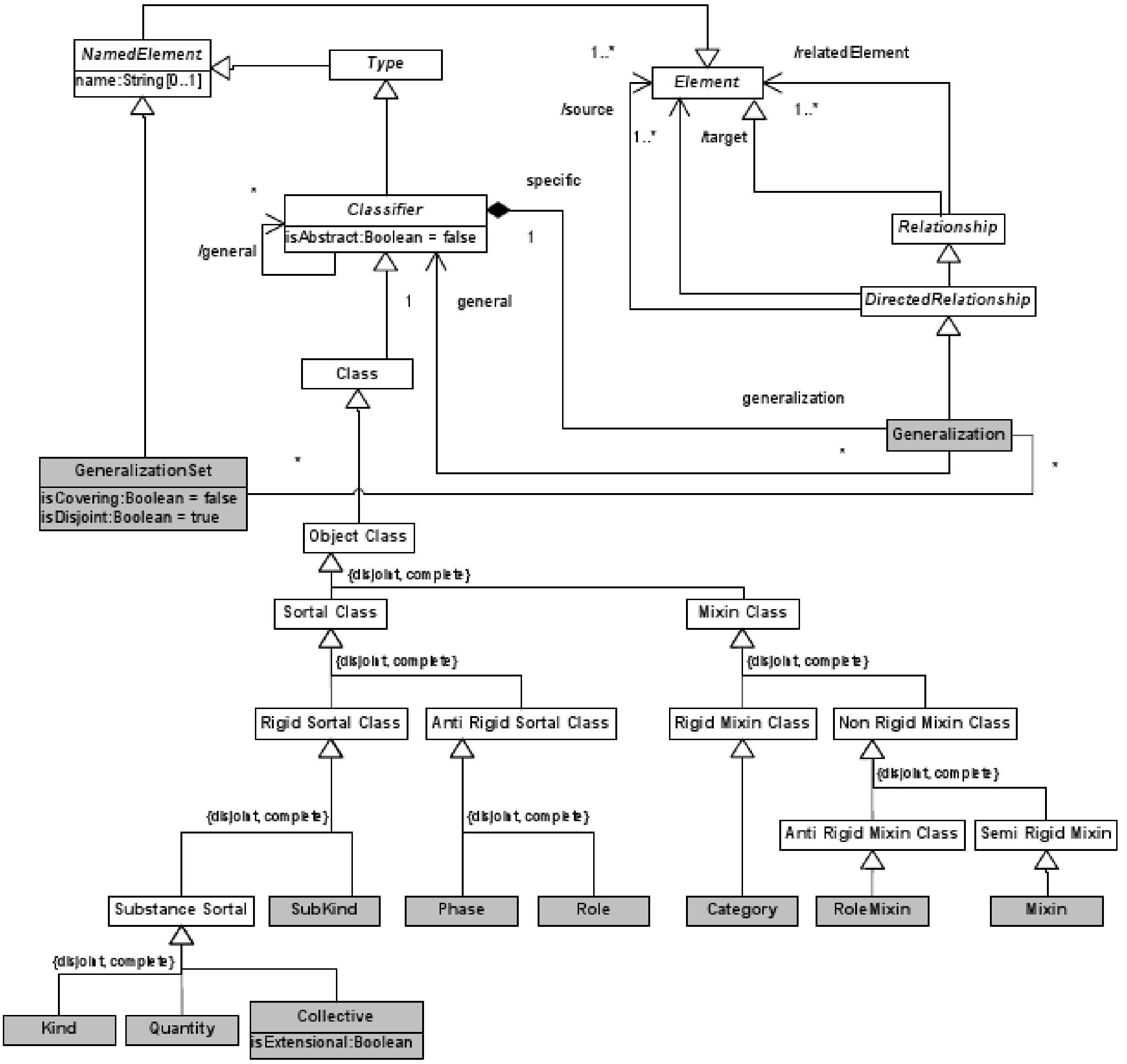


Figure 9: Revised fragment of the UML 2.0 metamodel according to the ontological categories of Fig. 7 (11, p. 316).

Description: *Substance Sortal* is an abstract metaclass that represents the general properties of all *substance sortals*, *i.e.*, rigid, relationally independent object universals that supply a principle of identity for their instances. Substance Sortal has no concrete syntax. Thus, symbolic representations are defined by each of its concrete subclasses.

Constraints:

1. Every substantial object represented in a conceptual model using this profile must be an instance of a substance sortal, directly or indirectly. This means that every concrete element of this profile used in a class diagram (isAbstract = false) must include in its

general collection one class stereotyped as either «kind», «quantity» or «collective»;

2. A substantial object represented in a conceptual model using this profile cannot be an instance of more than one ultimate substance sortal. This means that any stereotyped class in this profile used in a class diagram must not include in its general collection more than one substance sortal class. Moreover, a substance sortal must also not include another substance sortal nor a «subkind» in its general collection, *i.e.*, a substance sortal cannot have as a supertype a member of {«kind», «subkind», «quantity», «collective»};

3. A Class representing a rigid substantial universal cannot be a subclass of a Class representing an anti-rigid universal. Thus, a substance sortal cannot have as a supertype (must not include in its general collection) a member of {«phase», «role», «roleMixin»}.

Stereotype: «collective»

Description: A «collective» represents a substance sortal whose instances are *collectives*, *i.e.*, they are collections of complexes that have a uniform structure. Examples include a deck of cards, a forest, a group of people, a pile of bricks. Collectives can typically relate to complexes via a constitution relation. For example, a pile of bricks that *constitutes* a wall, a group of people that *constitutes* a football team. In this case, the collectives typically have an extensional principle of identity, in contrast to the complexes they constitute. For instance, The Beatles was in a given world w constituted by the collective {John, Paul, George, Pete} and in another world w' constituted by the collective {John, Paul, George, Ringo}. The replacement of Pete Best by Ringo Star does not alter the identity of the *band*, but creates a numerically different *group of people*.

Constraints:

1. A collective can be extensional. In this case the meta-attribute *isExtensional* is equal to *True*. This means that all its parts are essential and the change (or destruction) of any of its parts terminates the existence of the collective. We use the symbol {extensional} to represent an extensional collective.

Stereotype: «subkind»

Description: A «subkind» is a rigid, relationally independent restriction of a substance sortal that carries the principle of identity supplied by it. An example could be the subkind MalePerson of the kind Person. In general, the stereotype «subkind» can be omitted in conceptual models without loss of clarity.

Constraints:

1. A «subkind» cannot have as a supertype (must not include in its general collection) a member of {«phase», «role», «roleMixin»}.

Stereotype: «phase»

Description: A «phase» represents the phased-sortals phase, *i.e.* anti-rigid and relationally independent universals defined as part of a partition of a substance sortal. For instance, ⟨Caterpillar, Butterfly⟩ partitions the kind Lepdopterum.

Constraints:

1. Phases are anti-rigid universals and, thus, a «phase» cannot appear in a conceptual model as a supertype of a rigid universal;
2. The phases $\{P_1 \ldots P_n\}$ that form a phase-partition of a substance sortal S are represented in a class diagram as a disjoint and complete generalization set. In other words, a GeneralizationSet with (isCovering = true) and (isDisjoint = true) is used in a representation mapping as the representation for the ontological concept of a phase-partition.

Stereotype: «role»

Description: A «role» represents a phased-sortal role, *i.e.* anti-rigid and relationally dependent universal. For instance, the role student is played by an instance of the kind Person.

Constraints:

1. Roles are anti-rigid universals and, thus, a «role» cannot appear in a conceptual model as a supertype of a rigid universal.

Metaclass: Mixin Class

Description: *Mixin Class* is an abstract metaclass that represents the general properties of all *mixins*, *i.e.*, non-sortals (or dispersive universals). Mixin Class has no concrete syntax. Thus, symbolic representations are defined by each of its concrete subclasses.

Constraints:

1. A class representing a non-sortal universal cannot be a subclass of a class representing a Sortal. As a consequence of this postulate we have that a mixin class cannot have as a supertype (must not include in its general collection) a member of {«kind», «quantity», «collective», «subkind», «phase», «role»};
2. A non-sortal cannot have direct instances. Therefore, a mixin class must always be depicted as an abstract class (isAbstract = true).

Stereotype: «category»

Description: A «category» represents a rigid and relationally independent *mixin*, *i.e.*, a dispersive universal that aggregates essential properties which are common to different substance sortals. For example, the category RationalEntity as a generalization of Person and IntelligentAgent.

Constraints:

1. A «category» cannot have a «roleMixin» as a supertype. In other words, together with condition 1 for all mixins we have that a «category» can only be subsumed by another «category» or a «mixin».

Stereotype: «mixin»

Description: A «mixin» represents properties which are essential to some of its instances and accidental to others (semi-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.

Constraints:

1. A «mixin» cannot have a «roleMixin» as a supertype.

3.2 Classifiers and Properties

The Fig. 10 represents the UFO's metamodel excerpt regarding Classifiers and Properties. In this excerpt, the metaclass Moment Universal (also present in the metamodel excerpt pictured in Fig. 7) is specialized in Intrinsic Moment Universals, which characterizes Object Universals and are existentially dependent on them, and Relator Universals, which are existentially dependent on many entities.

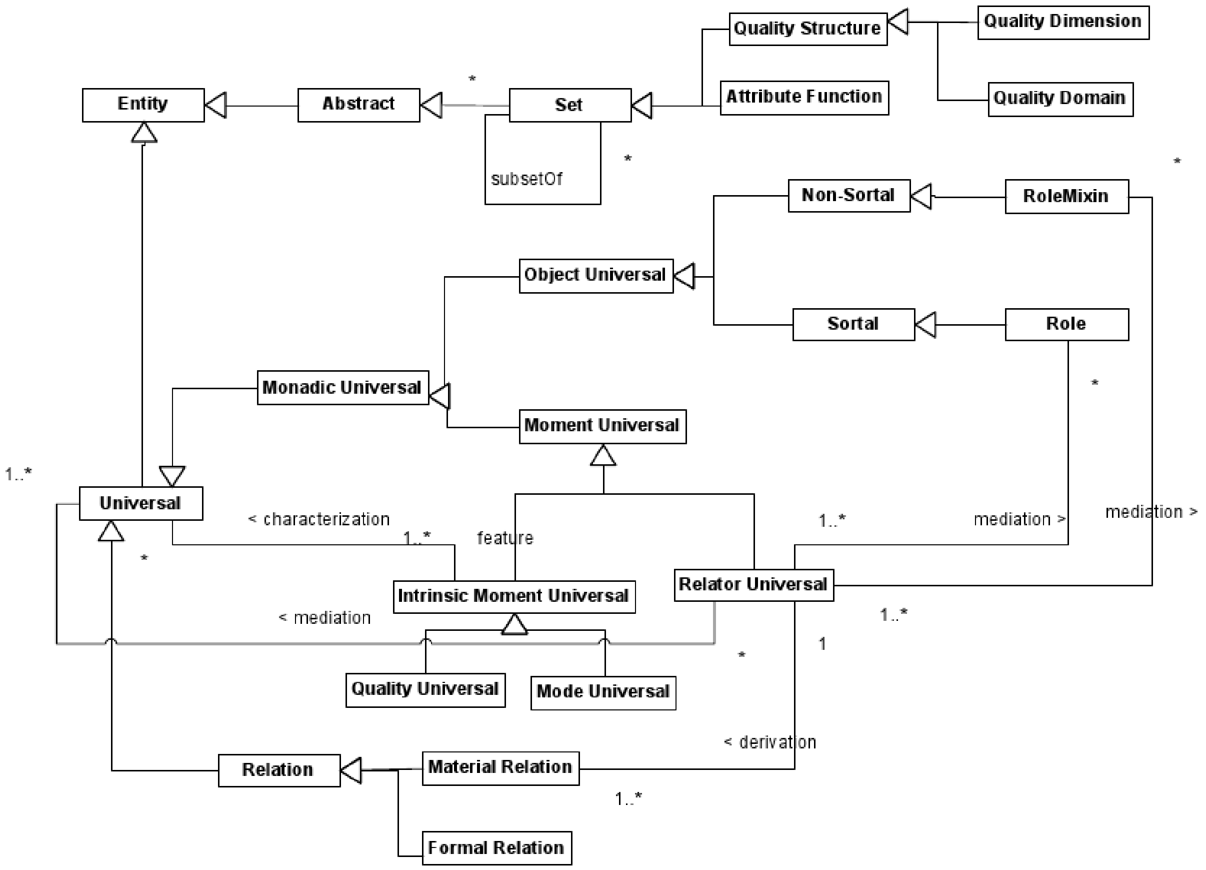


Figure 10: UFO excerpt regarding Relations, Moments, Quality Structures and related categories (11, p. 324).

Quality Structures are spaces of values in which individual qualities can obtain their values. The concept of Quality Structures represents "the ontological interpretation of the UML DataType construct"(11, p. 326). This concept is specialized into Quality Dimensions and Quality Domains. A Quality Dimension is composed of the set of values that a quality can take and the formal relations between them. A Quality Domain is a set of Quality Dimensions[4]. For example, the type of quality "color" is associated to a threedimensional Quality Domain composed by the Quality Dimensions hue, saturation and brightness, so every color has as a value a point in a threedimensional Quality Domain.

The instances of Moment Universal can only exist in other individuals, by *inhering* on them. The inherence relation of x in y is symbolized as $i(x,y)$[5] and implies existential dependence (see Definition 1) from the instances of Moment Universals to other individuals, named their bearers (11, p. 213). **Definition 10 (Bearer of a Moment)**: The bearer of a moment x is the unique[6] individual y such that x inheres in y. Formally, $\beta(x) \triangleq \iota y\ i(x,y)$. ■ (11, p. 214). Existential dependence can be used to differentiate Intrinsic Moment Universals and Relator Universals: instances of Intrinsic Moment Universals are dependent on a single individual; instances of Relator Universals depend on a plurality of individuals (11, p. 213).

Intrinsic Moment Universals are the foundation for attributes and formal comparative relationships. Intrinsic Moment Universal is specialized into Quality Universal and Mode Universal. A Quality Universal is an instance of Intrinsic Moment Universal that is associated to a Quality Structure (11, p. 224) by means of Attribute Functions. A Mode Universal is a Intrinsic Moment Universal that is not directly related to Quality Structures (11, p. 237) (*e.g.*, abilities, beliefs and thoughts are existentially dependent on a single Person).

Attribute Functions are used to map instances of Quality Universals to points in Quality Structures. "…attribute functions are therefore the ontological interpretation of UML attributes, *i.e.*, UML Properties which are owned by a given classifier.". (11, p. 325)[7]

In order to exemplify Quality Dimensions, Quality Domains, Quality Universals and Attribute Functions, let us create an OntoUML model, pictured in Fig. 11, in which we have the Substantial Universal Person (actually, a Kind), whose instances exemplify the Quality Universal Age (in other words, the Kind Person has an "age" attribute). Thus, for an arbitrary instance x of Person there is a quality a (instance of the Quality Universal Age) that *inheres* in x. Associated with Age, and in the context of a given measurement system, there is a Quality

[4] In OntoUML, Quality Dimensions are represented by Simple Datatypes, and Quality Domains are represented by Structured Datatypes. Although SimpleDatatypes and StructuredDatatypes were not present in Fig. 15 (taken from 11, p. 334), they are present in the OntoUML metamodel as specializations of the metaclass Datatype.

[5] See 11, p. 213 for the definition of inherence

[6] The upside-down iota operator (ι) used in a formula such as $\iota x\phi$ is the definite description operator defined by Whitehead and Russel in 57, p. 181, implying both the existence and the uniqueness of an individual x satisfying predicate ϕ.

[7] In OntoUML, an Attribute Function is represented by a Datatype Relationship from the owning Classifier to a Datatype, such that its navigable end name is the name of the attribute. Although Datatype Relationships were not present in Fig. 15, created in the specification of OntoUML in 11, they are present at later revisions of the language as specializations of the metaclass Directed Binary Relationship.

Dimension *ageValue* that is a linear structure isomorphic to the natural numbers ($age \in \mathbb{N}$) obeying the same ordering structure. Thus, we represented the Quality Dimension *ageValue* as the Simple Datatype NaturalNumber shown in Fig. 11. In this case, we can define an Attribute Function *age(Year)*, which maps each instance of Person (and in particular x) onto a point in the Quality Dimension *ageValue* (in OntoUML, we represented the Attribute Function *age(Year)* as a Datatype Relationship with "age" as its navigable end name, from the Kind Person to the Simple Datatype NaturalNumber). In a similar way, we state that the instances of Substantial Universal Person also exemplify the Quality Universal Birthday (in other words, the Kind Person has a "birthday" attribute). Associated with Birthday, there is a Quality Domain *birthdayValue* that contains three Quality Dimensions, namely, *dayValue*, *monthValue* and *yearValue* that are natural numbers (we are simplifying the calendar domain here, because for each calendar system (*e.g.*, Gregorian, Julian, Hebrew, Buddhist, Hindu, Persian, Islamic, Chinese, Ethiopian, *etc*), days, months and years may have different constraints on their possible values[8]. For example, for the Gregorian calendar, we have to model another field for "A.D" or "B.C." statements, or model years as integers instead of natural numbers). Therefore, we represented the Quality Domain *birthdayValue* as the Structured Datatype Birthday, which has three Datatype Relationships (with "day", "month" and "year" as the navigable end names) to the Simple Datatype NaturalNumber. In this case, we can define an Attribute Function *birthday*, which maps each instance of Person (and in particular x) onto a vector in the corresponding 3-dimensional Quality Domain *birthdayValue* (in OntoUML, we represented the Attribute Function *birthday* as a Datatype Relationship with "birthday" as its navigable end name, from the Kind Person to the Structured Datatype Birthday).

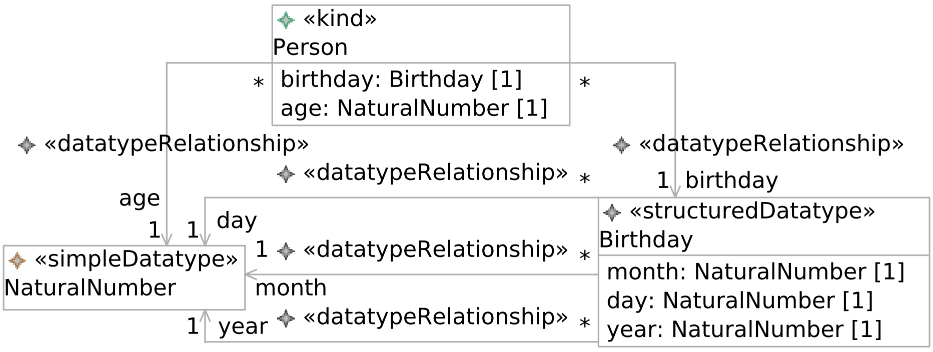


Figure 11: Example of Quality Structures.

An Intrinsic Moment Universal IMU is said to characterize an Universal U if for every x that is an instance of the Universal U (symbolized as x::U) there is at least one y::IMU such that $i(y,x)$ (11, p. 221). For example, the Mode Universal Mental State characterizes the Kind

[8] After choosing a calendar system, those constraints could be written as OCL expressions in the model shown in Fig. 11.

Person, so every instance x of Person bearers (for bearing, see Definition 10) at least one instance y of Mental State, as shown in Fig. 12.

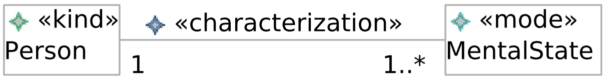


Figure 12: Example of Characterization.

Relator Universals are aggregations of all *qua individuals* that share the same foundation (11, p. 240). A *qua individual* is defined as an individual that bears all externally dependent modes (Mode Universals' instances) of an entity that share the same dependencies and the same foundation. External dependence is defined in Definition 11 (11, pp. 218,238) in the following way:

> **Definition 11 (Externally Dependent Mode)**: A mode x is externally dependent iff it is existentially dependent of an individual which is independent of its bearer (see Definition 10). Formally, $ExtDepMode(x) \triangleq Mode(x) \land \exists y\; indep(y, \beta(x)) \land ed(x,y)^{9}$. ■ (11, p. 238)
> **Definition 12 (Independence)**: $indep(x,y) \triangleq \neg ed(x,y) \land \neg ed(y,x)$. ■ (11, p. 218)

Intuitively, a *qua individual* is the consideration of an individual only w.r.t certain aspects that it has due to the participation in a certain relation (*e.g.*, Alex qua student, due to Alex being enrolled in a School) (11, p. 239).

A Relator Universal RU mediates another Substantial Universal SU by means of a mediation relation, symbolized by $mediation(SU, RU)$. If x, y and z are three distinct individuals such that: (i) x::RU; (ii) y is a *qua individual* and y is a *part of* x; (iii) z::SU and y inheres in z; then we have that x mediates z, symbolized by $m(x,z)$ and such that m::*mediation* (11, pp. 240-241). Also, every instance of a Relator Universal must mediate at least two disjoint instances of Substantial Universals. Relator Universals are the foundation for Material Relations.

The metaclass Relation is specialized into Material Relation and Formal Relation. Formal Relations are relations derived from intrinsic properties of the related entities. A Formal Relation "hold between two or more entities directly without any further intervening individual"(11, p. 236) (*e.g.*, the Formal Relation olderThan, derived from the property age of the type Person; instantiation (::); inherence (i) and existential dependence (ed)). Contrariwise, Material Relations are dependent on extrinsic relationships: the relata of a Material Relation must be mediated by individuals that are Relator Universals (11, p. 236).

[9] ed(x,y) is defined in Definition 1.

In general, a Relation R can be then be formally defined by the following schema:

Definition 13 (Formal and Material Relations): Let $\varphi(a_1,\ldots,a_n)$ denote a condition on the individuals $a_1,\ldots,a_n$. A Relation R is defined for the Universals $\mathrm{U}_1,\ldots,\mathrm{U}_n$ iff

$$\forall a_1,\ldots,a_n(\mathrm{R}(a_1,\ldots,a_n) \leftrightarrow ((\bigwedge\nolimits_{i\leq n}\mathrm{U}_i(a_i)) \wedge \varphi(a_1,\ldots,a_n)))$$

A Relation is a Material Relation if there is a Relator Universal U_R such that the condition φ is obtained from U_R as follows:

$$\varphi(a_1,\ldots,a_n) \leftrightarrow \exists k(\mathrm{U}_R(k)\wedge\bigwedge\nolimits_{i\leq n}m(k,a_i))$$

In this case, we say that the relation R is derived from the relator universal $\mathrm{U_R}$, or symbolically, $derivation(\mathrm{R},\mathrm{U_R})$. Otherwise, if such a Relator Universal $\mathrm{U_R}$ does not exists, R is termed a Formal Relation. ■

We have then that a n-tuple $(a_1,\ldots,a_n)$ instantiates a Material Relation R iff there is one relator r (instance of U_R) that mediates (and is existentially dependent on) every single a_i.

The derivation relationship between a Relator Universal and a Material Relation is a specialized relationship that indicates how instances of Material Relations can be derived from instances of mediation relations. This relation of derivation between a Material Relation and a Relator Universal is represented in OntoUML by the symbol ●– – – – – – –, in which the black circle is connected to the Relator Universal.

In order to exemplify Relator Universals, Formal Relations, Material Relations, mediations and derivations, let us create an OntoUML model, pictured in Fig. 13, in which we have that a Kind Person has an "age" attribute that is a natural number represented by the Quality Dimension Age(Year), which leads to the existence of an olderThan Formal Relation. Also, an instance of Person can play the role of a Student when he/she is enrolled in a School by means of a mediation relation (m) holding from an instance of the Relator Universal Enrollment to him/her. In a similar way, a Kind Organization can play the role of a School when it provides educational services by means of a mediation relation (m) holding from an instance of the Relator Universal Enrollment to it. When the same instance x of Enrollment is mediating an instance y of Person (*i.e.*, $m(x,y)$) and an instance z of Organization (*i.e.*, $m(x,z)$), then there is a Material Relation study between y and z. This fact is symbolized by the derivation relation between the Material Relation study and the Relator Universal Enrollment.

Again, by applying the framework for assessment of modeling languages proposed in 11, pp. 28-36 and using the excerpt of UFO shown in Fig. 10, 11, p. 323 revise the excerpt of the UML 2.0 metamodel shown in Fig. 14, creating the excerpt of the OntoUML metamodel pictured

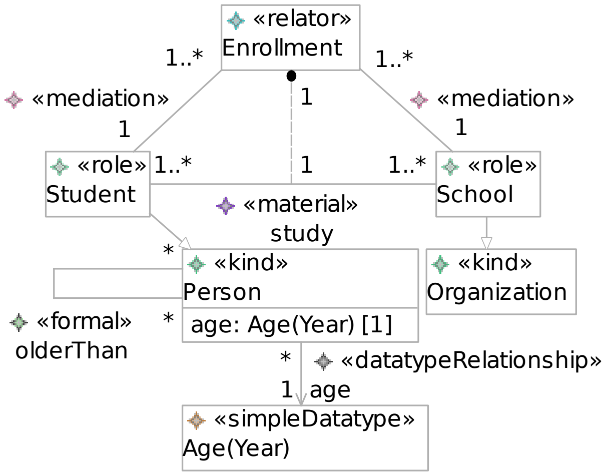


Figure 13: Example of Relator Universal, Formal Relation, Material Relation, mediation and derivation.

by Fig. 15.

OntoUML profile regarding the categories depicted in Fig. 15

Stereotype: «role»

Constraints:

2. Every «role» class must be connected to an association end of a «mediation» relation.

Stereotype: «roleMixin»

Description: A «roleMixin» represents an anti-rigid and externally dependent non-sortal, *i.e.*, a dispersive universal that aggregates properties which are common to different roles. In includes formal roles such as whole and part, and initiatior and responder.

Constraints:

1. Every «roleMixin» class must be connected to an association end of a «mediation» relation.

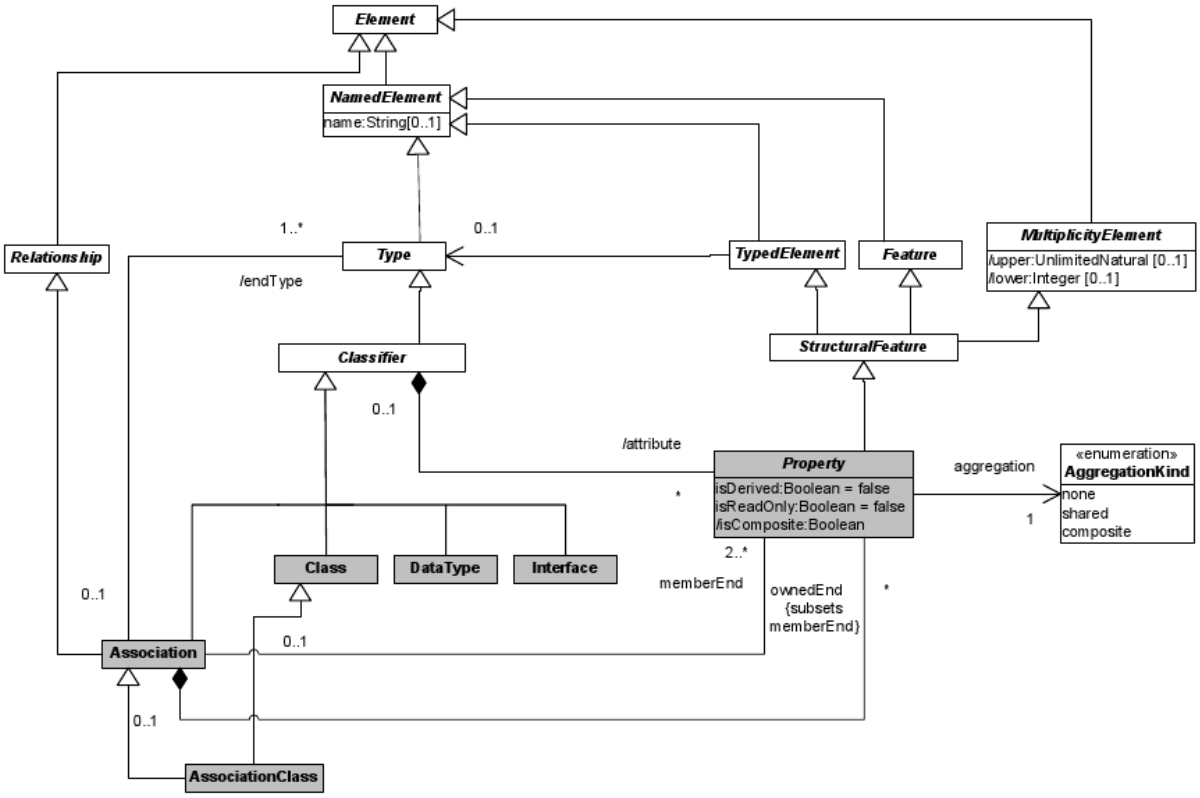


Figure 14: Excerpt of the UML metamodel featuring classifiers and Properties (11, p. 321).

Stereotype: «mode»

Description: A «mode» universal is an intrinsic moment universal. Every instance of mode universal is existentially dependent of exactly one entity. Examples include skills, thoughts, beliefs, intentions, symptoms, private goals.

Constraints:

1. Every «mode» must be (directly or indirectly) connected to an association end of at least one «characterization» relation.

Stereotype: «relator»

Description: A «relator» universal is a relational moment universal. Every instance of relator universal is existentially dependent of at least two distinct entities. Relators are the instantiation of relational properties such as marriages, kisses, handshakes, commitments, and purchases.

Constraints:

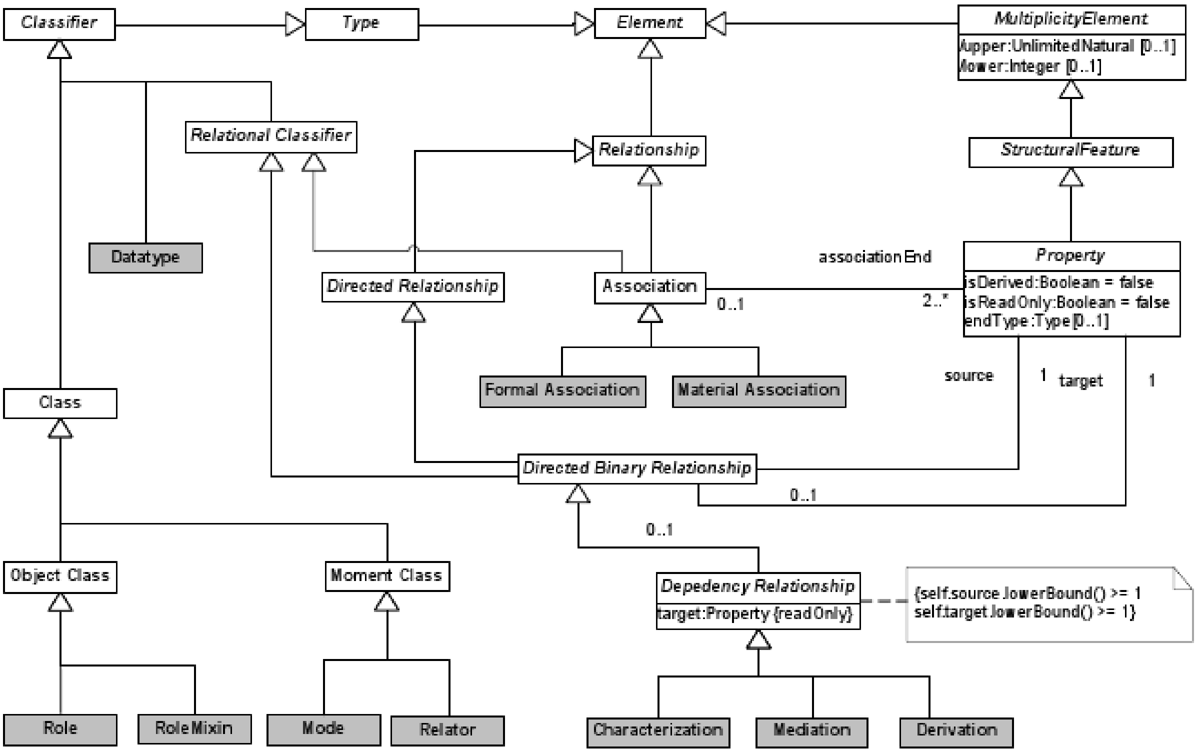


Figure 15: Revised fragment of the UML 2.0 metamodel according to the ontological categories of Fig. 10 (11, p. 334).

1. Every «relator» must be (directly or indirectly) connected to an association end on at least one «mediation» relation;

2. Let R be a relator universal and let $\{C_1 \ldots C_n\}$ be a set of universals mediated by R (related to R via a «mediation» relation). Finally, let lower_{C_i} be the value of the minimum cardinality constraint of the association end connected to C_i in the «mediation» relation. Then, we have that $(\sum_{i=1}^{n} \text{lower}_{C_i}) \geq 2$.

Stereotype: «mediation»

Description: A «mediation» is a formal relation that takes place between a relator universal and the endurant universal(s) it mediates. For example, the universal Marriage mediates the role universals Husband and Wife, the universal Enrollment mediates Student and University, and the universal Covalent Bond mediates the universal Atom.

Constraints:

1. An association stereotyped as «mediation» must have in its source association end a class

stereotyped as «relator» representing the corresponding relator universal (self.source. oclIsTypeOf(Relator)=true);

2. The association end connected to the mediated universal must have the minimum cardinality constraints of at least one (self.target.lower $\geq$ 1);
3. The association end connected to the mediated universal must have the property (self.target.isReadOnly = true);
4. The association end connected to the relator universal must have the minimum cardinality constraints of at least one (self.source.lower $\geq$ 1);
5. «mediation» associations are always binary associations.

Stereotype: «characterization»

Description: A «characterization» is a formal relation that takes place between a mode universal and the endurant universal this mode universal characterizes. For example, the universals Private Goal and Capability characterize the universal Agent.

Constraints:

1. An association stereotyped as «characterization» must have in its source association end a class stereotyped as «mode» representing the characterizing mode universal (self.source.oclIsTypeOf(Mode)=true);
2. The association end connected to the characterized universal must have the cardinality constraints of one and exactly one (self.target.lower = 1 and self.target.upper = 1);
3. The association end connected to the characterizing quality universal (source association end) must have the minimum cardinality constraints of one (self.source.lower $\geq$ 1);
4. The association end connected to the characterized universal must have the property (self.target.isReadOnly = true);
5. «characterization» associations are always binary associations.

Stereotype: Derivation Relation

Description: A derivation relation represents the formal relation of derivation that takes place between a material relation and the relator universal this material relation is derived from. Examples include the material relation married-to, which is derived from the relator universal Marriage, the material relation kissed-by, derived from the relator universal Kiss, and the material relation purchases-from, derived from the relator universal Purchase.

Constraints:

1. A derivation relation must have one of its association ends connected to a relator universal (the black circle end) and the other one connected to a material relation (self.target.oclIsTypeOf(Relator)=true, self.source.oclIsTypeOf(Material Association)=true);
2. Derivation associations are always binary associations;
3. The black circle end of the derivation relation must have the cardinality constraints of one and exactly one (self.target.lower = 1 and self.target.upper = 1);
4. The black circle end of the derivation relation must have the property (self.target. isReadOnly = true);
5. The cardinality constraints of the association end connected to the material relation in a derivation relation are a product of the cardinality constraints of the «mediation» relations of the relator universal that this material relation derives from. This is done in the manner shown in subsection 4.3.3. However, since «mediation» relations require a minimum cardinality of one on both of its association ends, then the minimum cardinality on the material relation end of a derivation relation must also be ≥ 1 (self.source.lower ≥ 1).

Stereotype: «material»

Description: A «material» association represents a material relation, *i.e.*, a relational universal which is induced by a relator universal. Examples include student *studies in* university, patient is *treated in* medical unit, person is *married to* person.

Constraints:

1. Every «material» association must be connected to the association end of exactly one derivation relation;
2. The cardinality constraints of the association ends of a material relation are derived from the cardinality constraints of the «mediation» relations of the relator universal that this material relation is derived from. This is done in the manner shown in subsection 4.3.3. However, since «mediation» relations require a minimum cardinality of one on both of its association ends, then the minimum cardinality constraint on each end of the derived material relation must also be ≥ 1;
3. Every «material» association must have the property (isDerived = true).

Metaclass: Property

Description: An attribute in the UML metamodel is a property owned by a classifier. Attributes are used in this profile to represent attribute functions derived for quality universals. Examples are the attributes color, age, and startingDate.

Constraints:

1. A property owned by a classifier (representing an attribute of that classifier) must have the minimum cardinality constraints of one (self.lower ≥ 1).

3.3 Aggregation and Composition

The Fig. 16 represents the UFO's metamodel excerpt regarding Aggregation and Composition. In this excerpt, the metaclass Entity (also present in the metamodel excerpt pictured in Fig. 10) has a partOf relation with itself. This relation is an anti-symmetric and non-transitive relation (*i.e.*, transitivity holds for certain cases but not for others), because two of its subclasses are transitive (subQuantityOf and subCollectionOf), one is intransitive (memberOf), and one is itself non-transitive (componentOf). Also, this relation obeys the irreflexivity axiom and weak supplementation principle (11, p. 342).

This partOf relation is of significant importance in conceptual modeling, being present in practically all conceptual modeling languages (*e.g.*, OPEN Modelling Language (9) (OML), UML, Enhanced Entity-Relationship (EER)). An important aspect to be addressed by any

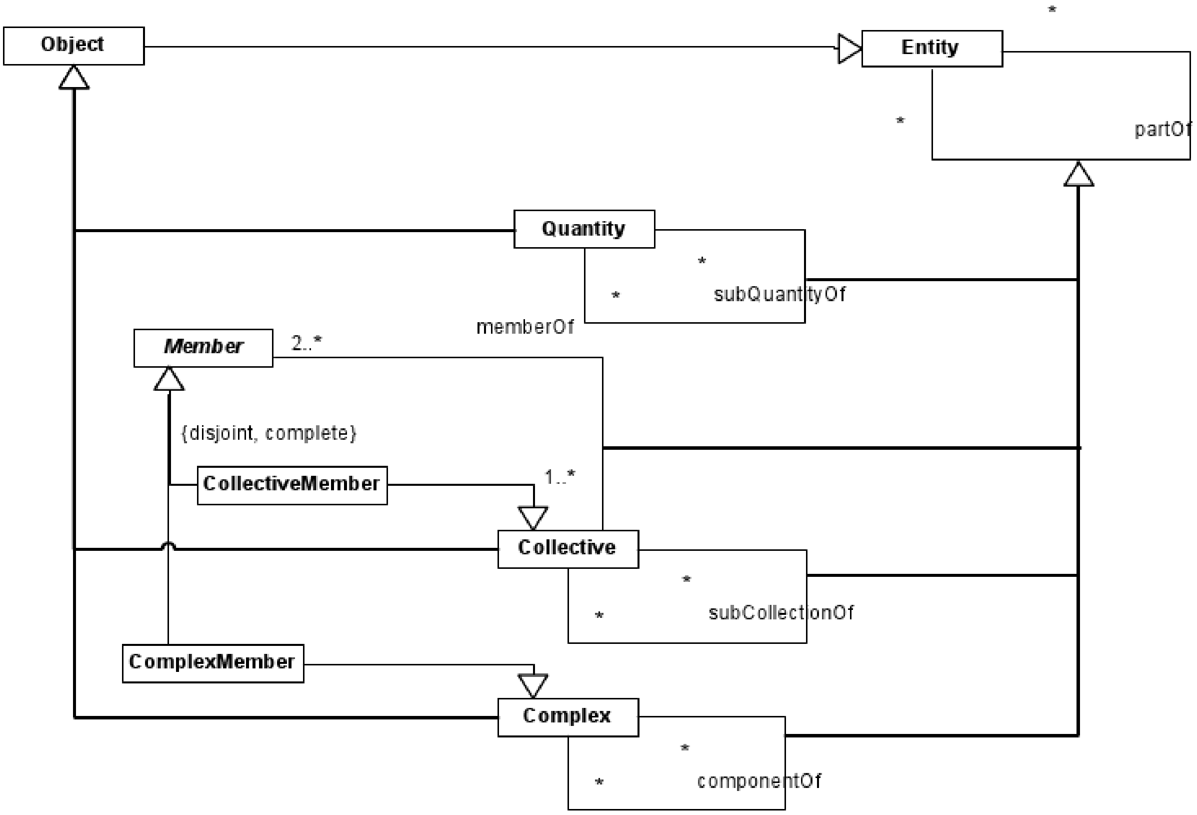


Figure 16: UFO excerpt regarding meronymic relations (11, p. 341).

conceptual theory of parthood is to stipulate the different status that parts can have w.r.t. the whole they compose. As discussed by (55), many of the issues regarding this point cannot be clarified without considering *modality*.

We can distinguish two types of part-whole relations based on the distinction between the previously defined notion Existential Dependence (Definition 1) and the one of Generic Dependence (Definition 14).

> **Definition 14 (generic dependence)**: An individual y is generically dependent on a type T iff, whenever y exists it is necessary that an instance of T exists. This can be formally characterized by the following *formula schema*: $GD(y,\mathrm{T}) \triangleq \Box(\varepsilon(y) \rightarrow \exists \mathrm{T}, x(\varepsilon(x)))$[10]. ■ (55, p. 12)

As one can observe contrasting the definitions 1 and 14, the former is a relation between two individuals, whilst the latter is a relation between an individual and a universal.

The essential parthood relations and the inseparable ones are relations that imply existential dependence. Contrariwise, a mandatory parthood relation is one that implies generic dependence

[10] This definition is formalized in a language of modal logics defined in 11 and in which all quantification is restricted by Sortals (11, pp. 121-122), so $\exists\mathrm{T},x(\mathrm{A})$ means that there is a x, taken from a set of instances of a SortalUniversal T, that satisfies A.

from the part to the whole (mandatory whole) or from the whole to the part (mandatory part). These types of parthood are defined in the sequel:

> **Definition 15 (essential part)**: An individual x is an essential part of another individual y iff, y is existentially dependent on x and x is, necessarily, a part of y: $EP(x,y) \triangleq ed(y,x) \wedge \Box(x \leq y)$. This is equivalent to stating that $EP(x,y) \triangleq \Box((\varepsilon(y) \rightarrow \varepsilon(x)) \wedge (x \leq y))$. We adopt here the *mereological continuism* defended by 58, which states that the part-whole relation should only be considered to hold among existents, *i.e.*, $\forall x,y((x \leq y) \rightarrow (\varepsilon(x) \wedge \varepsilon(y)))$. As a consequence, we can have this definition in its final simplification: $EP(x,y) \triangleq \Box(\varepsilon(y) \rightarrow (x \leq y))$. ■(55, p. 11)
> **Definition 16 (inseparable part)**: An individual x is an inseparable part of another individual y iff, x is existentially dependent on y, and x is, necessarily, a part of y: $IP(x,y) \triangleq \Box(\varepsilon(x) \rightarrow (x \leq y))$. ■ (55, p. 14)
> **Definition 17 (mandatory part)**: An individual x is a mandatory part of another individual y iff, y is generically dependent of a type T that x instantiates, and y has, necessarily, as a part an instance of T: $MP(\mathrm{T},y) \triangleq \Box(\varepsilon(y) \rightarrow \exists \mathrm{T},x(x < y))$. ■ (55, p. 12)
> **Definition 18 (mandatory whole)**: An individual y is a mandatory whole for another individual x iff, x is generically dependent on a type T that y instantiates, and x is, necessarily, part of an individual instantiating T: $MW(\mathrm{T},x) \triangleq \Box(\varepsilon(x) \rightarrow \exists \mathrm{T},y(x < y))$. ■ (55, p. 14)

Also, 11, p. 286 and 55, p. 17 suggest that when the whole is anti-rigid, we shall call the part immutable instead of essential. As every essential part is also immutable (11, p. 286)(55, pp. 17-18), then essential part is a specialization of immutable part. While immutable parts hold for rigid, semi-rigid or anti-rigid wholes, essential parts are maintained only between a part and a rigid whole.

Therefore, regarding modality, the parthood relations can be divided into two non-disjoint groups:

i The relations between individuals, such as relations that are essential (Definition 15), inseparable (Definition 16) or immutable;

ii The relations between types, such as relations that are mandatory regarding the part (Definition 17) or mandatory regarding the whole (Definition 18)

Furthermore, the part-whole relationships can also be divided into four disjoint groups, regarding the types of its domains: (i) componentOf relationships; (ii) subQuantityOf relationships; (iii) subCollectionOf relationships; and (iv) memberOf relationships. In the sequel, we will describe each type.

The metaclasses Quantity and Collective are the same metaclasses shown in Fig. 7 and explained in section 3.1. The metaclass Complex represents the functional complexes, which are instances that are composed by parts that play a multitude of roles in the context of the whole, differently from instances of Collectives (11, p. 187).

The componentOf parthood relation is a relation between two functional complexes. “Examples include: (a) my hand is part of my arm; (b) a car engine is part of a car; (c) an Arithmetic and Logic Unit (ALU) is part of a CPU; (d) a heart is part of a circulatory system.” (11, p. 350). ComponentOf relations are non-reflexive anti-symmetric non-transitive parthood relations, which obey the weak supplementation principle (11, p. 350).

The subQuantityOf parthood relation is a relation between two Quantities. “Examples include: (a) alcohol is part of Wine; (b) Plasma is part of Blood; (c) Sugar is part of Ice Cream; (d) Milk is part of Cappuccino.” (11, p. 350). SubQuantityOf relations are essential (see Definition 15) non-shareable (see Definition 20) non-reflexive anti-symmetric transitive parthood relations, which obey the strong supplementation principle[11] and the extensionality principle[12] (11, p. 350).

The subCollectionOf parthood relation is a relation between two Collectives. “Like quantities, collectives are maximal entities. However, in contrast with quantities, the unifying relation of a collective is not necessarily one of physical connection. For this reason, a collective can be shared by two or more collectives.” (11, p. 346) “Examples include: (a) the north part of the Black Forest is part of the Black Forest; (b) The collection of Jokers in a deck of cards is part of that deck of cards; (c) the collection of forks in cutlery set is part of that cutlery set; (d) the collection of male individuals in a crowd is part of that crowd.” (11, p. 351). SubCollectionOf relations are non-reflexive anti-symmetric transitive parthood relations, which obey the weak supplementation principle (11, p. 351).

The memberOf parthood relation is a relation between a singular functional complex or a Collective (as a part) and a Collective (as a whole). “Examples include: (a) a tree is part of forest; (b) a card is part of a deck of cards; (c) a fork is part of cutlery set; (d) a club member is part of a club.” (11, p. 352). MemberOf relations are non-reflexive anti-symmetric intransitive parthood relations, which obey the weak supplementation principle (11, p. 352).

Moreover, all parthood relations can be characterized regarding their shareability. 13, p. 39 is intentionally vague when stating that:

> “Precise semantics of shared aggregation varies by application area and modeler.

[11] In simple words, the strong supplementation principle states that if an individual y is not a part of another individual x then there is a part of y that does not overlap with x (11, p. 146).

[12] In a nutshell, the extensionality principle states that two objects are identical iff they have the same (proper) parts. This is the mereological counterpart of the extensionality principle in set theory, which states that two sets are identical iff they have the same members (11, p. 147).

> The order and way in which part instances are created is not defined." (13, p. 39)

From 11, p. 162, we have definitions for non-shareable (exclusive) parts (see Definition 19) and for general exclusive part-whole relations (see Definition 20).

> **Definition 19 (exclusive part)**: An individual x of type A is said to be an exclusive (proper) part of another individual y of type B (symbolized as $<_X (x, \mathrm{A}, y, \mathrm{B})$) iff y is the only B that has x as part.
> $<_X(x, \mathrm{A}, y, \mathrm{B}) \triangleq (x :: \mathrm{A}) \wedge (y :: \mathrm{B}) \wedge (x < y) \wedge (\forall z\ (z :: \mathrm{B})\ (x < z) \rightarrow (y = z))$ ■ (11, p. 162)
> **Definition 20 (general exclusive part-whole relation)**: A universal A is related to a universal B by a relation of general exclusive parthood (symbolized as $\mathrm{A} <_{GX} \mathrm{B}$) iff every instance x of A has an exclusive part of type B.
> $\mathrm{A} <_{GX} \mathrm{B} \triangleq \forall x\ (x :: \mathrm{A}) \rightarrow \exists y\ (y :: \mathrm{B}) \wedge <_X(x, \mathrm{A}, y, \mathrm{B})$
> or simply,
> $\mathrm{A} <_{GX} \mathrm{B} \triangleq \forall x\ (x :: \mathrm{A}) \rightarrow \exists! y\ (y :: \mathrm{B}) \wedge (x < y)$ ■ (11, p. 162)

Finally, by applying the framework for assessment of modeling languages proposed in 11, pp. 28-36 and using the excerpt of UFO shown in Fig. 16, 11, p. 341 revise the excerpt of the UML 2.0 metamodel shown in Fig. 14, creating the excerpt of the OntoUML metamodel pictured by Fig. 17.

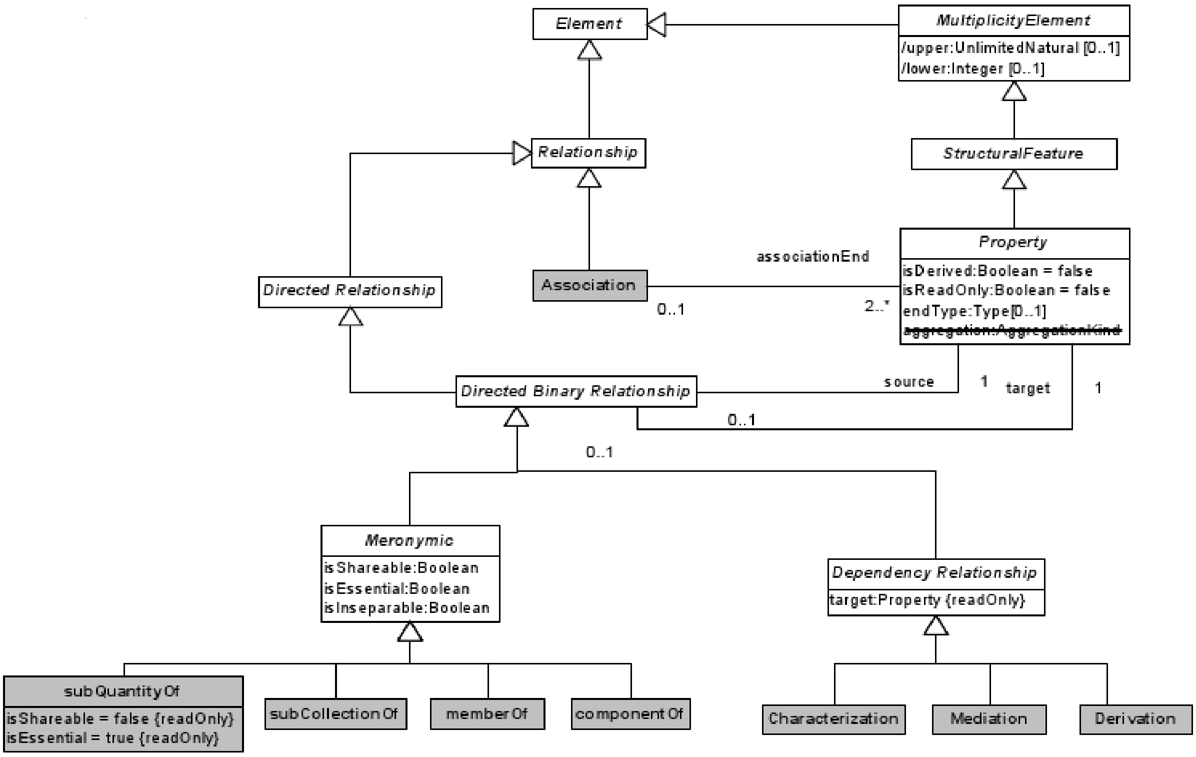


Figure 17: Revised fragment of the UML 2.0 metamodel according to the ontological categories of Fig. 16 (11, p. 348).

OntoUML profile regarding the categories depicted in Fig. 17

Metaclass: Meronymic

Description: Abstract metaclass representing the general properties of all meronymic relations. Meronymic has no concrete syntax. Thus, symbolic representations are defined by each of its concrete subclasses.

Constraints:

1. Weak supplementation: Let U be a universal whose instances are wholes and let $\{C_1 \ldots C_n\}$ be a set of universals related to U via aggregation relations. Let lower_{C_i} be the value of the minimum cardinality constraint of the association end connected to C_i in the aggregation relation. Then, we have that $(\sum_{i=1}^{n} \text{lower}_{C_i}) \geq 2$;
2. Essential Parthood: The *isEssential* attribute represents whether the meronymic relation is one of essential parthood, *i.e.*, whether the part is essential to the whole. In case the classifier connected to the association end representing the whole is an anti-rigid classifier, then the meta-attribute *isEssential* must be false, whereas the meta-attribute *isImmutable* may be true. However, if *isEssential* is true (in case of a rigid classifier with essential parts) then *isImmutable* must also be true. The concrete representation of this meta-property is via the tagged value essential decorating the association;
3. Inseparable Parthood: The *isInseparable* attribute represents whether the meronymic relation is one of inseparable parthood, *i.e.*, whether the whole is essential to the part. The concrete representation of this meta-property is via the tagged value {inseparable} decorating the association;
4. Shareable Parthood: The *isShareable* attribute represents whether the meronymic relation is (locally) shareable, *i.e.*, whether the part can be related to more than a whole of that kind. The concrete representation of this meta-property is via the color property of the symbol used to depict this relation (a diamond with or without a decorating letter): if (isShareable = true) then the symbol is shown in white color, otherwise, it is shown in black.

Metaclass: componentOf

Description: componentOf is a parthood relation between two complexes. Examples include: (a) my hand is part of my arm; (b) a car engine is part of a car; (c) an ALU is part of a CPU; (d) a heart is part of a circulatory system.

Meta-properties: Non-reflexivity, Anti-Symmetry, Non-Transitivity and Weak Supplementation.

Constraints:

1. The classes connected to both association ends of this relation must represent universals whose instances are functional complexes. A universal X is a universal whose instances are functional complexes if it satisfies the following conditions: (i) If X is a sortal universal, then it must be either stereotyped as «kind» or be a subtype of a class stereotyped as «kind»; (ii) Otherwise, if X is a mixin universal, then for all classes Y such that Y is a subtype of X, we have that Y cannot be either stereotyped as «quantity» or «collective», and Y cannot be a subtype of class stereotyped as either «quantity» or «collective».

Metaclass: subQuantityOf

Description: subQuantityOf is a parthood relation between two quantities. Examples include: (a) alcohol is part of Wine; (b) Plasma is part of Blood; (c) Sugar is part of Ice Cream; (d) Milk is part of Cappuccino.

Meta-properties: Non-reflexivity, Anti-Symmetry, Transitivity and Strong Supplementation (Extensional Mereology).

Constraints:

1. This relation is always non-shareable (isShareable = false);

2. All entities stereotyped as «quantity» are extensional individuals and, thus, all parthood relations involving quantities are essential parthood relations;

3. The maximum cardinality constraint in the association end connected to the part must be one (self.target.upper = 1);

4. The classes connected to both association ends of this relation must represent universals whose instances are quantities. A universal X is a universal whose instances are quantities if it satisfies the following conditions: (i) If X is a sortal universal, then it must be either stereotyped as «quantity» or be a subtype of a class stereotyped as «quantity»; (ii) Otherwise, if X is a mixin universal, then for all classes Y such that Y is a subtype of X, we have that Y cannot be either stereotyped as «kind» or «collective», and Y cannot be a subtype of class stereotyped as either «kind» or «collective».

Metaclass: subCollectionOf

Description: subCollectionOf is a parthood relation between two collectives. Examples include: (a) the north part of the Black Forest is part of the Black Forest; (b) The collection of Jokers in a deck of cards is part of that deck of cards; (c) the collection of forks in cutlery set is part of that cutlery set; (d) the collection of male individuals in a crowd is part of that crowd.

Meta-properties: Non-reflexivity, Anti-Symmetry, Transitivity and Weak Supplementation (Minimum Mereology).

Constraints:

1. The classes connected to both association ends of this relation must represent universals whose instances are collectives. A universal X is a universal whose instances are collectives if it satisfies the following conditions: (i) If X is a sortal universal, then it must be either stereotyped as «collective» or be a subtype of a class stereotyped as «collective»; (ii) Otherwise, if X is a mixin universal, then for all classes Y such that Y is a subtype of X, we have that Y cannot be either stereotyped as «kind» or «quantity», and Y cannot be a subtype of class stereotyped as either «kind» or «quantity»;
2. The maximum cardinality constraint in the association end connected to the part must be one (self.target.upper = 1).

Metaclass: memberOf

Description: memberOf is a parthood relation between a complex or a collective (as a part) and a collective (as a whole). Examples include: (a) a tree is part of forest; (b) a card is part of a deck of cards; (c) a fork is part of cutlery set; (d) a club member is part of a club.

Meta-properties: Non-reflexivity, Anti-Symmetry, Intransitivity and Weak Supplementation. Although transitivity does not hold across two memberOf relations, a memberOf relation followed by subCollectionOf is transitive. That is, for all a,b,c, if memberOf(a,b) and memberOf(b,c) then ¬memberOf(a,c), but if memberOf(a,b) and subCollectionOf(b,c) then memberOf(a,c).

Constraints:

1. This relation can only represent essential parthood (isEssential = true) if the object representing the whole on this relation is an extensional (isExtensional = true) individual. In this case, all parthood relations in which this individual participates as a whole are essential parthood relations;

2. The classifier connected to association end relative to the whole individual must be a universal whose instances are collectives. The classifier connected to the association end relative to the part can be either a universal whose instances are collectives, or a universal whose instances are functional complexes.

4 A Tool for Building and Verifying OntoUML Models

In this chapter, we present an Eclipse-based graphical editor, which aims at fulfilling the absence of tool supporting for the OntoUML language. By representing UFO's categories and axiomatization in the language metamodel, the complexity of these foundational issues is hidden from the user while still constraining him to produce ontologically sound models.

In the following sections, we present the implementation of the OntoUML abstract syntax, syntactical constraints and concrete syntax proposed in 11 by using MDA technologies, in particular, the Ecore (23) and OCL languages for the abstract syntax, the OCL for the syntactical constraints, and the GMF Eclipse plug-in for the concrete syntax. Moreover, we also present the description of the whole set of metamodel transformations that leads to the implementation of the graphical editor.

This chapter is structured in the following way: Section 4.1 introduces the architecture of the graphical editor. Section 4.2 shows the Ecore metamodel that we created from the three fragments of the OntoUML metamodel pictured in Figs. 9, 15 and 17 in order to define the abstract syntax of the OntoUML language by using the Eclipse Modeling Framework[1] (3) (EMF) plug-in. Section 4.3 shows the implementation of the OntoUML syntactical constraints (or well-formedness rules) taken from 11, pp. 317–320, 334–338, 348–352 as OCL expressions. The section 4.4 shows the definition of the OntoUML concrete syntax (as specified in 11, pp. 317–320, 334–338, 348–352) by using the Graphical Modeling Framework[2] (GMF) plug-in. In the section 4.5, we show the set of transformations from the Ecore metamodel (embedded with OCL constraints), which leads to the creation of the Java code that implements the OntoUML graphical editor. In the section 4.6, we build a case study by means of a running example in order to show the capabilities of the editor. Finally, in section 4.7 we pose our conclusions for this chapter.

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

[2] http://www.eclipse.org/modeling/gmf.

4.1 Architecture

The architecture of the editor presented here is pictured in Fig. 18. This architecture has been conceived to follow a model-driven approach. In particular, we adopt the OMG MOF metamodeling architecture.

We define a metamodel (also called abstract syntax) for a language $\mathcal{L}$ as a conjuntion of (i) a taxonomy of $\mathcal{L}$'s concepts and (ii) a set of $\mathcal{L}$'s syntactic constraints, as shown in Fig. 18. In order to write the OntoUML Taxonomy, we use a MOF compliant language named Ecore. In Fig. 18, we represent the codification of the OntoUML Taxonomy in Ecore as the class Domain Model. Aside from Ecore, we also use the language OCL in the formalization of the OntoUML Metamodel. We use OCL expressions mainly to: (i) define how derived associations get their values; (ii) define how some meta-attributes get their initial values when belonging to instances of specific metaclasses (details are given in subsection 4.3.3); (iii) specify query operations and specify invariants, *i.e.*, integrity constraints that determine a condition that must be true in all consistent system states. Observe that (i) relates to the OntoUML Taxonomy and is specified within the Domain Model in Fig. 18, while (ii) and (iii) relate to the Set of OntoUML Syntactic Constraints and are specified within the classes OCL Expression for the Automatic Initialization of Meta-Attributes and OCL Invariant, respectively, which are part of the class Mapping Model.

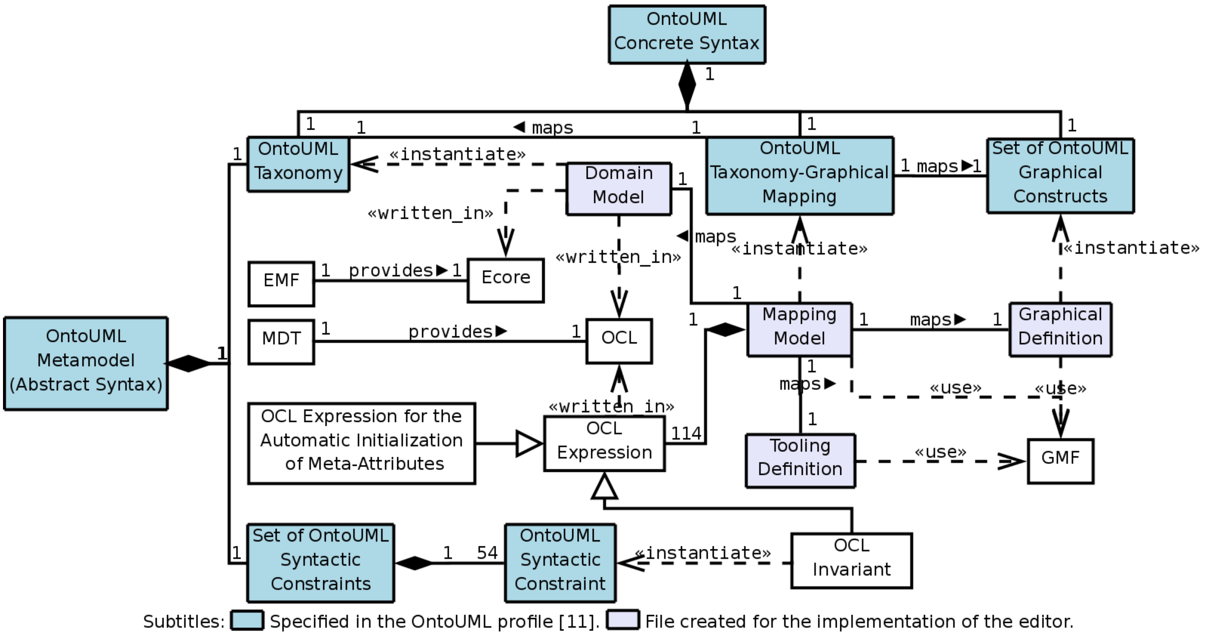

OntoUML Concrete Syntax
1
1
1
◄ maps
1
1
1
OntoUML Taxonomy
1
«instantiate»
Domain Model
1
OntoUML Taxonomy-Graphical Mapping
1 maps►1
Set of OntoUML Graphical Constructs
«written_in»
◄ maps
«instantiate»
«instantiate»
EMF
1 provides►1
Ecore
«written_in»
1
MDT
1
provides►
1
OCL
1
Mapping Model
1
maps►
1
Graphical Definition
OntoUML Metamodel (Abstract Syntax)
1
1
maps►
«use»
«use»
«written_in»
1
OCL Expression for the Automatic Initialization of Meta-Attributes
OCL Expression
114
Tooling Definition
«use»
GMF
OCL Invariant
Set of OntoUML Syntactic Constraints
1
1
54
OntoUML Syntactic Constraint
«instantiate»
Subtitles: 🟦 Specified in the OntoUML profile [11]. 🟪 File created for the implementation of the editor.

Figure 18: The architecture of the editor.

In terms of implementation technology, the editor is implemented by using a number of plug-ins that supports graphical editor development in the context of the Eclipse IDE. For the creation of the OntoUML Taxonomy in Ecore (the Domain Model), we have used EMF. The

EMF together with MDT allows for the creation and verification of Ecore models which have embedded OCL expressions.

We also define a concrete syntax for a language $\mathcal{L}$ as a conjuntion of (i) a taxonomy of $\mathcal{L}$'s concepts; (ii) a set of $\mathcal{L}$'s graphical constructs, *i.e.*, a set of the visual elements allowed by $\mathcal{L}$; and (iii) a mapping between $\mathcal{L}$'s taxonomy and $\mathcal{L}$'s graphical constructs; as shown in Fig. 18. In order to encode the OntoUML Concrete Syntax and to build the graphical interface of the editor, we have used EMF and GMF. The GMF provides a high-level description of visual representations to support transformation to a set of Java classes for the graphical editor using a Model-View-Controller (MVC) architecture. In Fig. 18 we illustrate some important artifacts that we created by means of the GMF plug-in: (i) the Graphical Definition specifies the Set of OntoUML Graphical Constructs; (ii) the Tooling Definition specifies a structure for the editor's toolbar; and (iii) the Mapping Model specifies the OntoUML Taxonomy-Graphical Mapping as well as a mapping between the OntoUML Taxonomy (as specified in the Domain Model), the Set of OntoUML Graphical Constructs (as specified in the Graphical Definition) and the toolbar specification (as specified in the Tooling Definition). Also, the Mapping Model contains the OCL Expressions for the Automatic Initialization of Meta-Attributes and, in order to make use of the GMF verification framework, this artifact is where the OCL Invariants that implement the Set of OntoUML Syntactic Constraints are defined.

Moreover, the Domain Model is concretely represented as a *.ecore file, the Graphical Definition as a *.gmfgraph file, the Tooling Definition as a *.gmftool file and the Mapping Model as a *.gmfmap file.

Finally, the Domain Model and the Mapping Model are sufficient (but not necessary[3]) for defining together the OntoUML Abstract Syntax; and the Domain Model, the Mapping Model and the Graphical Definition are sufficient (but not necessary[4]) for defining together the OntoUML Concrete Syntax.

4.2 Definition of the OntoUML Abstract Syntax in Ecore

In order to be able to use the EMF plug-in to transform the abstract syntax (or metamodel) of OntoUML to a set of Java classes that can hold a model in a MVC architecture, we define the OntoUML's metamodel in a language named Ecore.

[3] Because the Mapping Model contains more than the OCL Invariants.

[4] Because the Mapping Model contains more than the OntoUML Taxonomy-Graphical Mapping specification and also relates to the Tooling Definition.

So, we use here the EMF plug-in to create an Ecore metamodel for OntoUML by unifying the three fragments of the OntoUML metamodel pictured in Figs. 9, 15 and 17 in the metamodel pictured in Fig. 19[5].

Additionaly, as the meta-attribute *isImmutable* of the metaclass Meronymic is defined only for parts in 11, p. 286 and 55, pp. 17-18, we modified the OntoUML metamodel in order to create a related meta-attribute regarding the immutability of wholes by replacing the meta-attribute *isImmutable* by the meta-attribute *isImmutablePart* and creating a new meta-attribute *isImmutableWhole* in the metaclass Meronymic. See Definitions 25 and 30 for *isImmutablePart* and *isImmutableWhole*, respectively.

We also create OCL expressions in order to indicate how the derived meta-attributes and meta-relations of the OntoUML metamodel get their values. For example, the derived meta-attribute "general" of the metaclass "Classifier" is the transitive closure of the generalization relation regarding the specialized class. In OCL, "general" can be defined as:

Listing 3: Derivation of the meta-attribute "general" of metaclass "Classifier".

```
context Classifier::general:Bag(Classifier)
derive: self.allSuperTypes()
context Element::allSuperTypes():Bag(Element)
body: if self.oclIsKindOf(Classifier) then (if self.
    oclAsType(Classifier).generalization->forAll(x | x.
    oclIsUndefined()) then Set{} else Set{self.oclAsType(
    Classifier).generalization->collect(x | x.general),
    self.oclAsType(Classifier).generalization->collect(x |
    if x.general.oclIsKindOf(Classifier) then x.general.
    allSuperTypes() else Set{} endif)}->flatten() endif)
    else Set{} endif
```

The derived meta-relation "relatedElement" of metaclass "Relationship" is the set of all the entities that are related by a relation, independently if it is an Association or a DirectedRelationship. This meta-relation can be defined in OCL as:

Listing 4: Derivation of the meta-relation "relatedElement" of metaclass "Relationship".

```
context Relationship::relatedElement:Bag(Element)
derive: if self.oclIsKindOf(Association) then self.
    oclAsType(Association).associationEnd else if self.
```

[5] Besides containing these fragments, this metamodel contains some auxiliary attributes and relationships (which contains an "aux" in their names) created due to GMF requirements.

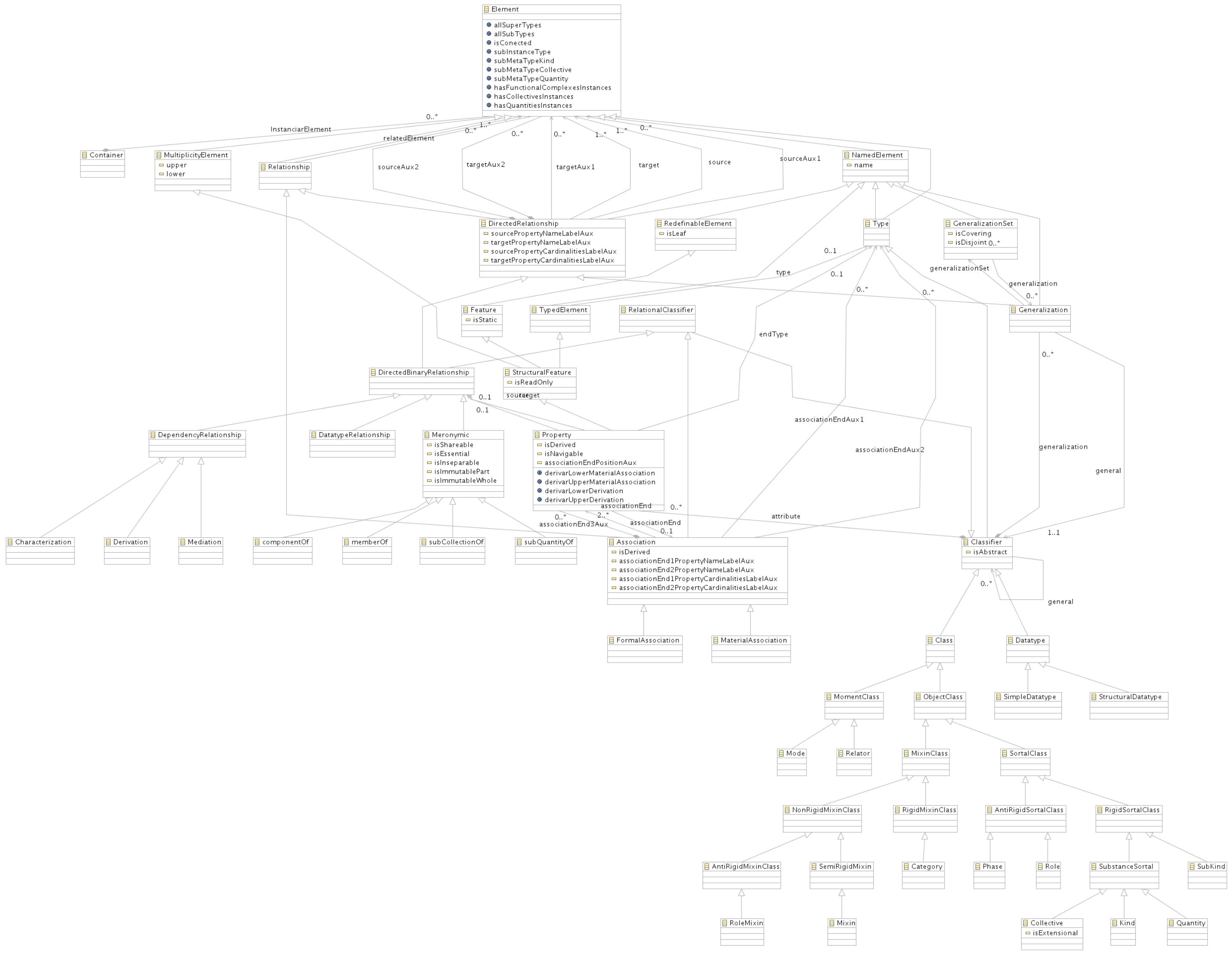


Figure 19: The Ecore OntoUML metamodel.

```
oclIsKindOf(DirectedRelationship) then Set{self.
oclAsType(DirectedRelationship).source, self.oclAsType(
DirectedRelationship).target}->flatten() else null
endif endif
```

The OCL expressions that we use to derive the complete set of derived meta-relations of the OntoUML metamodel are shown in sections A.3 and A.4 of appendix A.

4.3 Mapping the OntoUML Syntactical Constraints into OCL Expressions

All the syntactical constraints of the OntoUML language are defined as OCL expressions in order to make use of the verification framework provided by the GMF Eclipse plug-in. Therefore, in subsection 4.3.1 we show, in general terms, how we map these constraints to OCL, in subsection 4.3.2 we show some additional invariants that we create and which are not defined in the OntoUML profile (11), and subsection 4.3.3 deals with the editor's capability of automatically initializing or modifying some meta-attributes' values.

4.3.1 Mapping the Original OntoUML Invariants into OCL Expressions

In general, there are two types of syntactical constraints regarding the way in which they are verified:

- constraints that prevents the user from performing modeling actions that would put the model into an inconsistent state in which the only possible fix would be to undo the original modeling action. These constraints are feasible to be automatically verified whenever the user tries to update a model, and are classified as *live validation* by the GMF verification framework. In order to avoid misunderstandings, we will classify them as *live verification* instead;

- constraints that prevents the user from leaving the model into an incomplete state in its final version, *i.e.* leaving the final version of the model in a state in which there are possible ways to fix it rather then undoing previous actions. These constraints are not feasible to be automatically verified whenever the user tries to update a model, because a modeling action may put the model in an incomplete state in which further modeling actions are required in order to the model be considered syntactically correct. Therefore, these constraints have to be manually verified after the user deems suitable and are classified as *batch validation* by the GMF verification framework. In order to avoid misunderstandings, we will classify them as *batch verification* instead.

In order to create a constraint in GMF, we have to specify the metaclass that will be the context of the constraint. In an OntoUML model, when you change a class or relationship *x* that is instance of a metaclass *X*, only the *live* constraints that have *X* as context will be verified. This is a limitation of the GMF verification framework that we have to take in consideration when writing *live* constraints. For this reason, the choice of the context metaclass and the OCL expression that encodes some OntoUML syntactical constraints may appear non-intuitive.

Furthermore, the OntoUML syntactical constraints that are *live* can be categorized in the following types:

- Constraints about which types of classifiers can be in the general collection of a determined Classifier, *e.g.*, no RigidSortal can have an AntiRigidSortal in its general collection. This constraint can be modeled in OCL as:

```
context Generalization
inv SubstanceSortalConstraint3: if self.specific.
    oclIsKindOf(SubstanceSortal) then not (self.general
    .oclIsKindOf(AntiRigidSortalClass) or self.general.
    oclIsKindOf(RoleMixin)) else true endif
```

- Constraints about how many classifiers of a specific type can be in the general collection of a determined Classifier, *e.g.*, no Classifier may have more than one SubstanceSortal in its general collection. We can model this constraint in OCL as:

```
context Generalization
inv SubstanceSortalConstraint2a: self.specific.
    allSubTypes()->including(self.specific)->forAll(x |
     x.allSuperTypes()->select(y | y.oclIsKindOf(
    SubstanceSortal))->size() <= 1)
context Element::allSuperTypes():Bag(Element)
body: if self.oclIsKindOf(Classifier) then (if self.
    oclAsType(Classifier).generalization->forAll(x | x.
    oclIsUndefined()) then Set{} else Set{self.
    oclAsType(Classifier).generalization->collect(x | x
    .general), self.oclAsType(Classifier).
    generalization->collect(x | if x.general.
    oclIsKindOf(Classifier) then x.general.
    allSuperTypes() else Set{} endif)}->flatten() endif
    ) else Set{} endif
context Element::allSubTypes():Bag(Element)
body: let generalizations : Set(Generalization) =
    Generalization.allInstances()->select(x | x.general
     = self) in (if self.oclIsKindOf(Classifier) then (
    if generalizations->forAll(y | y.oclIsUndefined())
    then Set{} else Set{generalizations->collect(y | y.
    specific), generalizations->collect(y | if y.
    specific.oclIsKindOf(Classifier) then y.specific.
    allSubTypes() else Set{} endif)}->flatten() endif)
    else Set{} endif)
```

- Constraints on the unchangeability of some values of determined meta-attributes of specific Classifiers. As the values of these meta-attributes are always initialized with the correct value, their unchangeability can be verified in *live* mode. An example is the unchangeability of the value “true” in the meta-attribute isAbstract of the MixinClasses. In OCL, this constraint can be modeled as:

```
context MixinClass
inv MixinClassConstraint2: self.isAbstract = true
```

- Constraints on the types of Classifiers that can be in the extremities of specific OntoUML relationships, *e.g.*, a Characterization relationship must have an instance of Mode in its source extremity. This constraint can be modeled in OCL as:

```
context Characterization
inv CharacterizationConstraint1: self.source->forAll(x
     | if x.oclIsKindOf(Property) then x.oclAsType(
    Property).endType.oclIsTypeOf(Mode) else false
    endif)
```

- Constraints on the cardinalities of specific types of OntoUML relationships that are initialized with suitable values, *e.g.*, for the Characterization relationships, the association end connected to the characterized universal must have the cardinality constraints of one and exactly one; and the constraints on the values of the cardinalities of Material Associations and Derivation relationships. As these cardinalities are automatically calculated in the moment of the creation of the relations, we have to prohibit the user from entering inconsistent values. We can model the first example in OCL as:

```
context Property
inv CharacterizationConstraint2: if self.target.
    oclIsKindOf(Characterization) then ((self.lower =
    1) and (self.upper = 1)) else true endif
```

- Constraints on the type of numbers that can appear as cardinality values. For example, the cardinalities must be cardinal numbers. More specifically, they must be natural numbers ($\mathbb{N}$) or the least cardinal infinite $\aleph_0$, which is represented as *[6]. In OCL, this constraint can be modeled as:

```
context MultiplicityElement
inv MultiplicityElementConstraint1: (self.lower >= 0)
    or (self.lower = -1)
context MultiplicityElement
inv MultiplicityElementConstraint2: (self.upper >= 0)
    or (self.upper = -1)
```

The OntoUML syntactical constraints that are *batch* can be categorized in the following types:

- Constraints about the existence of Classifiers that must be in the general collection of a determined Classifier, *e.g.*, every non-abstract (isAbstract=false) non-SubstanceSortal must have a SubstanceSortal as its supertype. We can model this constraint in OCL as:

```
context ObjectClass
inv SubstanceSortalConstraint1: if ((self.isAbstract =
     false) and not self.oclIsKindOf(SubstanceSortal))
    then self.allSuperTypes()->exists(x | x.oclIsKindOf
    (SubstanceSortal)) else true endif
```

- Constraints about the existence of covering and disjoint GeneralizationSets relating the Phases of a Sortal, because the Phases must partition a Sortal. This constraint can be modeled in OCL as:

[6] In order to get * cardinalities, one shall enter the number -1.

```
context Phase
inv PhaseConstraint2: let general_substance_sortal :
    SubstanceSortal = Generalization.allInstances()->
    select(x | (x.specific = self) and (x.general.
    oclIsKindOf(SubstanceSortal)))->collect(x | x.
    general.oclAsType(SubstanceSortal))->any(true) in (
    let phase_generalizations : Set(Generalization) =
    Generalization.allInstances()->select(x | (x.
    general = general_substance_sortal) and (x.specific
    .oclIsKindOf(Phase))) in (let
    phase_generalization_sets : Set(GeneralizationSet)
    = GeneralizationSet.allInstances()->select(x | x.
    generalization->includesAll(phase_generalizations))
     in (if (general_substance_sortal.oclIsUndefined()
    or (phase_generalizations->size() = 1)) then true
    else ((phase_generalization_sets->size() = 1) and (
    phase_generalization_sets->forAll(x | (x.isCovering
     = true) and (x.isDisjoint = true)))) endif)))
```

- Constraints about the relational dependence (see Definition 22), *e.g.*, the relational dependence of (i) Roles, RoleMixins and Relators with Mediations relationships; (ii) Modes with Characterizations relationships; and (iii) the Material Associations with Derivation relationships. We can model the constraint regarding Roles in OCL as:

```
context Role
inv RoleConstraint2: Mediation.allInstances()->exists(
    x | x.target->exists(y | if y.oclIsKindOf(Property)
     then ((y.oclAsType(Property).endType = self) or (
    self.allSuperTypes()->includes(y.oclAsType(Property
    ).endType))) else false endif))
context Element::allSuperTypes():Bag(Element)
body: if self.oclIsKindOf(Classifier) then (if self.
    oclAsType(Classifier).generalization->forAll(x | x.
    oclIsUndefined()) then Set{} else Set{self.
    oclAsType(Classifier).generalization->collect(x | x
    .general), self.oclAsType(Classifier).
    generalization->collect(x | if x.general.
     oclIsKindOf(Classifier) then x.general.
     allSuperTypes() else Set{} endif)}->flatten() endif
     ) else Set{} endif
```

- Constraints on the cardinalities of specific types of OntoUML relationships, which can be set by the modeler but must obey certain rules, like the Weak Supplementation Axiom, which, in simple words, states that a whole must have at least two disjoint parts. In OCL, this constraint can be modeled as:

```
context Meronymic
inv MeronymicConstraint1: Meronymic.allInstances()->
     select(x | x.source->exists(y | if y.oclIsKindOf(
     Property) then (self.source->exists(z | if z.
     oclIsKindOf(Property) then (z.oclAsType(Property).
     endType = y.oclAsType(Property).endType) else false
      endif)) else false endif))->collect(w | w.target->
     collect(k | if k.oclIsKindOf(Property) then (if (k.
     oclAsType(Property).lower = -1) then 2 else k.
     oclAsType(Property).lower endif) else 0 endif)->sum
     ())->sum() >= 2
```

Subsection 4.3.3 will extend the categories of live and batch constraints when explaining the automatic initialization or modification of meta-attributes which values are constrained by the values of other meta-attributes.

The whole set of OCL expressions that we use to define these syntactical constraints is shown in sections A.1 and A.3 of appendix A.

4.3.2 Some Additional Invariants for OntoUML

This subsection documents a set of OCL invariants that are not related to the OntoUML syntactical constraints as documented in the OntoUML profile (11, pp. 317–320, 334–338, 348–352). We create these invariants and formalize them in OCL (see sections A.2 and A.3 of appendix A for the OCL formalization) as part of the OntoUML syntactical constraints because we think they are important in the specification of OntoUML models.

Some of these invariants were not specified in the OntoUML profile because they are similar to invariants taken from the UML metamodel. For example, the invariants that state that the interval specified by the minimum and maximum cardinalities must be a non-empty (possibly infinite) set of natural numbers (*i.e.*, the minimum cardinality must be less or equal than the maximum cardinality) are present in the UML metamodel (13, p. 95, constraint 2). As, we implemented the OntoUML metamodel from scratch, we have to specify these UML invariants in OCL.

Additional Invariants

- In all association ends of the associations stereotyped as «formal» or «material», the maximum cardinality constraint (the property upper) must be equal or greater to the lower cardinality constraint (the property lower);
- A relationship stereotyped as «datatypeRelationship» must have in its target association end a class stereotyped as «simpleDatatype» or «structuredDatatype», because attributes represent attribute functions derived for quality universals;
- A relationship stereotyped as «datatypeRelationship» must have in its source association end an instance of Classifier, excepting instances of «simpleDatatype», because a «simpleDatatype» is a Datatype that has no attributes. Excepting for Generalizations, GeneralizationSets and Properties, all the other OntoUML constructs are Classifiers;
- Every relationship stereotyped as «datatypeRelationship» that have a «structuredDatatype» in its source association end must have the meta-attribute isReadOnly = true in its target association end. This constraint came from (13, p. 61), which basically states that an instance of a «structuredDatatype» cannot change its attributes without ceasing to be the same instance:

> “All copies of an instance of a data type and any instances of that data type with the same value are considered to be the same instance. Instances of a data type that have attributes (*i.e.*, is a structured data type) are considered to be the same if the structure is the same and the values of the corresponding attributes are the same. If a data type has attributes, then instances of that data type will contain attribute values matching the attributes.”. (13, p. 61)

- An association stereotyped as «characterization», «mediation», «componentOf», «memberOf», «subCollectionOf», «subQuantityOf» or a derivation relation must have its source extremity connected to only one element;
- An association stereotyped as «characterization», «mediation», «componentOf», «memberOf», «subCollectionOf», «subQuantityOf» or a derivation relation must have its target extremity connected to only one element;
- An association stereotyped as «characterization», «mediation», «componentOf», «memberOf», «subCollectionOf», «subQuantityOf» or a derivation relation must have its source extremity connected to a Property;
- An association stereotyped as «characterization», «mediation», «componentOf», «memberOf», «subCollectionOf», «subQuantityOf» or a derivation relation must have its target extremity connected to a Property;
- In the source of the relationships stereotyped «characterization», «mediation», «componentOf», «memberOf», «subCollectionOf», «subQuantityOf» or a derivation relation, the maximum cardinality constraint (the property upper) must be equal or greater to the lower cardinality constraint (the property lower);
- In the target of the relationships stereotyped «characterization», «mediation», «componentOf», «memberOf», «subCollectionOf», «subQuantityOf» or a derivation relation, the maximum cardinality constraint (the property upper) must be equal or greater to the lower cardinality constraint (the property lower);
- A generalization must have its source extremity connected to at maximum one element;
- A generalization must have its target extremity connected to at maximum one element;
- An association stereotyped as «material» must be connected to Properties which have its meta-attributes isDerived = true;
- The maximum cardinalities of an association stereotyped as «material» are calculated automatically;
- The cardinalities on the source extremity of a derivation relationship are calculated automatically.
- There cannot be two «mediation» relationships x and y having the same ground, *i.e.*, $domain(x) = domain(y)$ and $codomain(x) = codomain(y)$;

- In a relationship stereotyped as «componentOf», «memberOf», «subCollectionOf» or «subQuantityOf» (*i.e.*, a Meronymic relationship), if the meta-attribute isImmutablePart = true, then the Properties connected on its target association ends must have the meta-attribute isReadOnly = true;

- In a relationship stereotyped as «componentOf», «memberOf», «subCollectionOf» or «subQuantityOf» (*i.e.*, a Meronymic relationship), if the meta-attribute isInseparable = true (in case of an anti-rigid class with inseparable parts), then the meta-attribute isImmutableWhole must also be true;

- In a relationship stereotyped as «componentOf», «memberOf», «subCollectionOf» or «subQuantityOf» (*i.e.*, a Meronymic relationship), if the meta-attribute isImmutableWhole = true, then the Properties connected on its source association ends must have the meta-attribute isReadOnly = true;

- Non-shareability implies a cardinality of exactly one in the extremity connected to the whole (see Definition 20);

- The minimum cardinality constraint (the meta-attribute lower) must be a natural number ($\mathbb{N}$) or the least cardinal infinite $\aleph_0$, which is represented as *;

- The maximum cardinality constraint (the meta-attribute upper) must be a natural number ($\mathbb{N}$) or the least cardinal infinite $\aleph_0$, which is represented as *;

- A class stereotyped as «structuredDatatype» must have at least two disjoint attributes (a class can have attributes by means of relationships stereotyped as «datatypeRelationship»);

- A relationship stereotyped as «subQuantityOf» must have the meta-attribute isImmutablePart = true, because these relationships are always essential (*i.e.*, the meta-attribute isEssential = true);

- In a relationship stereotyped as «subQuantityOf», the Properties related to the parts must have the meta-attribute isReadOnly = true, because these parts are always immutable (*i.e.*, the meta-attribute isImmutablePart = true).

The whole set of OCL expressions that we use to define those invariants is shown in sections A.2 and A.3 of appendix A.

4.3.3 Automatic Initialization or Modification of Meta-Attributes' Values

In this subsection, we present the meta-attributes which values can be automatically initialized or modified by the OntoUML Editor. We will classify these meta-attributes into two overlapping categories:

1. The ones for which there are invariants constraining them to have specific values as long as their owners exist. For example, the second syntactical constraint for the metaclass MixinClass, shown in page 39 and formalized in OCL in Listing 27, states that for Categories, Mixins and RoleMixins, the meta-attribute isAbstract must have the value "true". For every meta-attribute belonging to this category, if the user has to set its value to the correct one (*i.e.*, the value that is syntactically correct) by hand, the related constraint must be a batch one. However, if its value is automatically set to the correct one by the editor, in the creation of its owner class or relation, the related constraint can be a live one, disallowing the user to change the value of the meta-attribute.

2. The ones for which there are invariants constraining them to have specific values as long as other determined meta-attributes are set to other specific values. In other words, a meta-attribute a of a metaclass X is in this category if there is another meta-attribute b of a metaclass Y, some values v_1 and v_2, and an invariant constraining an instance x of X to have the value v_1 in a as long as a determined instance y of Y has the value v_2 in b (possibly, $X = Y$, $a = b$, $v_1 = v_2$ or $x = y$). For example, the second syntactical constraint for the metaclass Collective, shown in page 37 and formalized in OCL in Listing 21, states that when a Collective C has the value "true" in its meta-attribute isExtensional, for all Meronymic relationships R that have C as the whole, R's meta-attribute isEssential must have the value "true". Again, in an instance x of X, if the user has to set a's value to v_1 by hand when he sets b's value to v_2 in y, the related constraint must be a batch one. However, if the editor automatically modify a's value to v_1 in x when the user sets b's value to v_2 in y and $X \neq Y$, then the related constraint can be a live one, disallowing the user to change a's value in x to an incorrect one as long as b's value in y is set to v_2. Otherwise, in case $X = Y$, then the related constraint must be a batch one, because the editor can only automatically modify the value of b in y to v_2 after the verification of the live constraint. In other words, when $X = Y$, the constraint must be batch, because if it was live, as the context of the constraint is X ($= Y$), it would be verified when the user tries to set b's value to v_2 in y, and would disallow the setting in the case of a's value in x be different than v_1. However, when $X \neq Y$, the constraint may be live, because, as X will be the context, the constraint will not be verified when the user changes b's value in y to v_2.

We automatically initialize meta-attributes belonging to the first category by means of the OCL language and the GMF plug-in, specifically, by editing the GMFMap file. We add a "Feature Seq Initializer" to select the metaclass of the meta-attribute, a "Feature Value Spec" in order to select the meta-attribute and a "Value Expression" to write the OCL expression that calculates the correct value for the meta-attribute.

However, in order to automatically modify the value of meta-attributes belonging to the second category, we have to manually customize the Java code generated by the GMF framework, ocasionally also making use of OCL expressions, as for the calculation of the values of the upper cardinality constraint for both association ends of Material Associations and the lower and upper cardinality constraints of the source association end of Derivation relationships.

By defining those categories of meta-attributes, we can extend our categories of live and batch constraints with the following ones:

New categories for live constraints:

- The constraints regarding meta-attributes belonging to the first category.
- The constraints regarding meta-attributes belonging to the second category and such that $X \neq Y$.

New category for batch constraints:

- The constraints regarding meta-attributes belonging to the second category and such that $X = Y$.

The meta-attributes in the first category are: isAbstract (of the metaclass Classifier), lower (of the metaclass MultiplicityElement), upper (of the metaclass MultiplicityElement), isReadOnly (of the metaclass StructuralFeature), isDerived (of the metaclass Association), isShareable (of the metaclass Meronymic), isEssential (of the metaclass Meronymic), isDerived (of the metaclass Property) and isImmutablePart (of the metaclass Meronymic).

The meta-attributes in the second category are: lower (of the metaclass MultiplicityElement), upper (of the metaclass MultiplicityElement), isEssential (of the metaclass Meronymic), isImmutablePart (of the metaclass Meronymic), isExtensional (of the metaclass Collective), isReadOnly (of the metaclass StructuralFeature) and isImmutableWhole (of the metaclass Meronymic).

The explanations of the reasons of the pertinence of each of those meta-attributes in those categories, as well as the OCL expressions utilized in order to initialize the value of those meta-attributes are given in section A.5 of appendix A.

Finally, when “false”, the isShareable meta-attribute of the metaclass Meronymic specifies that the cardinality in the association end connected to the whole (the source association end) must be exactly one, *i.e.*, both values of the lower and upper meta-attributes of the Property in the source association end must be “1” (Definition 20, guaranteed by the fourth additional syntactical constraint for the metaclass Meronymic, shown in page 72 and formalized in OCL in Listing 84). The values of those cardinalities are not automatically modified by the editor when the user sets the value of isShareable to “false”, because it is possible that the cardinalities are correct, while the relationship is not actually non-shareable in the sense of Definition 20, *e.g.*, in case the user is confused by the ambiguous UML definition of shareability.

4.4 Definition of the OntoUML Concrete Syntax by using GMF

In this chapter, we discuss how the OntoUML concrete syntax (defined in the OntoUML profile (11, pp. 317–320, 334–338, 348–352)) is mapped in the GMF technologies for the creation of the graphical editor.

As discussed in the subsubsection 2.2.5.2, the concrete syntax of a language is mapped to GMF by a model (the *.gmfgraph file) that describes the appearance of the concrete syntax of the language.

In this model, the elements in the concrete syntax can be represented by using four different types of basic graphical elements within the GMF: Node, Connection, Compartment and Label. Nodes are polygons or custom figures; Connections are lines that connects a source object to a target object; Compartments are spaces inside Nodes in which one can put some graphical objects; and Labels are objects that contains text and that are linked to Nodes or Connections by a semi-visible link. By semi-visibility, we mean that an object is semi-visible iff it is only visible when selected or moving.

Nodes and Connections can assume various geometrical forms. However, in our work, we use just a few forms:

- Nodes are always represented as visible or semi-visible rectangles;
- Connections are represented as solid or dashed lines, which may have a decoration on its

source or target. We use five different decorations: a black circle; a black diamond; an empty diamond; an open arrow; and a closed arrow.

In summary, we model the OntoUML concrete syntax in the following way:

1. The metaclasses Mode, Relator, Category, Mixin, RoleMixin, Collective, Kind, Quantity, Phase, Role, simpleDatatype and structuredDatatype are represented as Nodes with a rectangular shape having a compartment for attributes (which is invisible when empty), a label «. . . » for their stereotypes, a label for the meta-attribute *name*, and, in the case of the metaclass Collective, a label {extensional}[7] for its meta-attribute *isExtensional*;

2. The metaclass GeneralizationSet is represented as a semi-visible Node with a rectangular shape having a label for the meta-attribute *name* and another label for the meta-attributes *isCovering* and *isDisjoint*;

3. The metaclasses Characterization, FormalAssociation and MaterialAssociation are represented as Connections with a solid line shape having a label «. . . » for their stereotypes, a label for its meta-attribute *name* and, in its extremities, labels for their role names and cardinalities (which get their values from the meta-attributes of the associated instances of Property);

4. The metaclass Mediation is represented as a Connection with a solid line shape having a label «mediation» for its stereotype, a label for its meta-attribute *name* and, in its extremities, labels for its cardinalities (which get their values from the meta-attributes of the associated instances of Property);

5. The metaclass Derivation is represented as a Connection with a dashed line shape decorated in the target with a black circle, and having, in its extremities, labels for its cardinalities (which get their values from the meta-attributes of the associated instances of Property);

6. The metaclasses componentOf, memberOf, subCollectionOf, subQuantityOf are represented as Connections with a solid line shape decorated in the source with a (black or empty, depending on its meta-attribute *isShareable*) diamond, and having a label «. . . » for their stereotypes, a label for the meta-attribute *name*, a label for the meta-attributes *isEssential*, *isInseparable*, *isImmutablePart* and *isImmutableWhole*[8], and, in its extremities, labels for

[7] In general, in a label, the text that is related to a boolean meta-attribute is only visible when the value of the meta-attribute is "true".

[8] This label has a distinct behaviour: when the value of *isEssential* is "true" then the value of *isImmutablePart* must be "true", so there is no need to show a label "{essential, immutable part}", because the label {essential} carries the same information. However, if the value of *isEssential* is "false" and the value of *isImmutablePart* is

its role names and cardinalities (which get their values from the meta-attributes of the associated instances of Property);

7. The metaclass Datatype Relationship is represented as a Connection with a solid line shape decorated in the target with an open arrow, and having a label «datatypeRelationship» for its stereotype, and, in its extremities, labels for its role names and cardinalities (which get their values from the meta-attributes of the associated instances of Property);

8. The metaclass Generalization is represented as a Connection with a solid line shape decorated in the source with a closed arrow;

9. The metaclass Property is represented as a label for all OntoUML Relationships (excepting Generalizations) or as an attribute in attribute compartments of the metaclasses that are connected with datatypes by Datatype Relationships.

Since OntoUML is an extension of UML, the concrete syntax of the former is basically the standard defined concrete syntax of latter. Even the additional modeling primitives and meta-attributes of OntoUML are modeled by the standard visualization of stereotypes and tagged values in UML, namely, names enclosed in guillemets (stereotypes) for modeling primitives and names enclosed in brackets for meta-attributes.

4.5 Transforming the Ecore Metamodel and Additional OCL Constraints in a Graphical Editor

In this section we describe how we implement the editor by using some Eclipse plug-ins. Subsection 4.5.1 shows how OntoUML Editor is implemented; subsection 4.5.2 deals with licensing the source code.

4.5.1 From the OntoUML Profile to OntoUML Editor

Firstly, we create an EMF project and build the Ecore metamodel depicted in Fig. 19 of section 4.2. In this metamodel, we also define some operations and some derived meta-references (which, in Ecore, are called EOperations and EReferences, respectively) by including the OCL formalizations posed in sections A.3 and A.4 of appendix A.

"true", then "immutable part" can no longer be suppressed from the label {immutable part}. A similar argument is valid for the meta-attributes *isInseparable* and *isImmutableWhole*.

Then, we automatically transform this Ecore metamodel into a Genmodel file. In this Genmodel, we set some variable names regarding the use of some JET templates, which are needed in order to enable the EMF plug-in to handle, by means of the MDT plug-in, the OCL expressions that are in the Ecore metamodel, as described in 49. From the Genmodel we perform two automatic transformations (following 50) and get the Java code for the Ecore metamodel and an EMF.Edit (51) framework, which provides generic reusable classes, which we use for building our graphical editor. We perform a number of customizations on the generated Java code, because, at the moment of writing this thesis, the EMF framework is not capable of generating code that implements the structure that we need for our editor. The source code is available at 59 (see section 4.5.2 for the licensing).

Thereon, we create a GMF project and perform more transformations on the Ecore metamodel. From the Ecore metamodel, we create a GMFGraph file, which describes the visualization of the graphical elements of the editor, namely, the types of nodes, the types and thickness of the lines that represents the OntoUML relationships and the decorations of the extremities of the relationships. In order to create this file, we indicate which metaclasses or meta-attributes will be the nodes, the connections or the labels. We create the OntoUML stereotypes as labels of the metaclasses that are defined as nodes.

From the Ecore metamodel, we also create a GMFTool file, in which, again, we select which metaclasses or meta-attributes will be the nodes, the connections or the labels. In this file, we manually create the names of the tools that will be in the tool palette of our editor and which are utilized in order to instantiate the OntoUML constructs (metaclasses or meta-attributes) in nodes or connections. We categorize these tools into three sections: Classes, Relationships and Rules.

In order to create a mapping between the Ecore metamodel, the GMFGraph file and the GMFTool file, we create a GMFMap file. In this file, we map the elements from the Ecore metamodel to their visualization specification (which is defined in the GMFGraph file) and their creation tools specification (which is defined in the GMFTool file). In order to create this file, we select which metaclass from the Ecore metamodel will be the root element. This root metaclass has to be related by Containment EReferences (*i.e.*, EReferences that have their “Containment” properties set to “true”) with every metaclass that shall be instantiable by the editor. Also, again we select which metaclasses or meta-attributes will be the nodes, the connections or the labels. In this GMFMap file, we also put the OCL constraints, which can be verified in live or in batch mode (see subsection 4.3.1 for live and batch constraints). In order to create live constraints, we put them inside “Audit Rules” that have their “Use In Live Mode” properties set to “true”. Otherwise, in order to create batch constraints, we put them inside “Audit Rules” that have their “Use In Live Mode” properties set to the default “false” value. One can see which OCL constraints are validated in live/batch mode in appendix A.

Once the changes in the GMFMap file are finished, we transform it to a GMFGen file. This file is utilized in the automatic generation of the Java code that implements the editor, by the GMF framework. In this file, we also set some properties in order to enable the validation of OCL constraints in OntoUML models: we set the properties “Live Validation UI Feedback”, “Validation Decorators” and “Validation Enabled” to “true”; and “Shortcuts Decorator Provider Priority” and “Validation Decorator Provider Priority” to “Highest”. Then, we generate the Java code that implements the OntoUML Editor.

Due to the Bug 138179 (60) (which is not fixed at the moment of writing this thesis), when we create an OntoUML relationship, we cannot automatically create the related Properties and their visualizations, as labels of the relationship. Therefore, in order to provide visualizations for cardinalities and role names of relationships, we had to include additional meta-attributes and meta-relations in the OntoUML metamodel as well as to customize the Java code generated by the EMF and GMF plug-ins.

4.5.2 Licensing

This graphical editor, named OntoUML Editor, is a Free and Open Source Software (FOSS) and its source code is licensed under GPLv3 (61) (see annex A) and, occasionally, EPLv1 (62) (see annex B). We consider GPLv3 more appropriate than EPLv1, because GPLv3 is a *strong copyleft* license, while EPLv1 is a *weak copyleft* one. Strong copyleft licenses are the licenses that must apply on all kinds of derived works, *e.g.*, all derived works from a GPLv3 licensed work must inherit its GPLv3 license. Unlike, weak copyleft licenses are the ones that allow some derived works to have different licenses (63).

The GPLv3 and EPLv1 licenses are not compatible. Therefore, if there is a part of the source code of OntoUML Editor that can be legally understood as a “derivative work” of some program licensed under the EPLv1 (*e.g.*, there is no clear understood if the code generated by some Eclipse plug-ins, for example, EMF and GMF, are considered to be a derivative work (64)), then, if accordingly to EPLv1 this part (or even the whole source code of OntoUML Editor) shall be considered as licensed under EPLv1, then we will consider it (or the whole source code of OntoUML Editor, if necessary) licensed under EPLv1. Otherwise, the source code of OntoUML Editor is licensed under GPLv3.

4.6 A Case Study for the OntoUML Editor: Building an OntoUML Model

In this section, we illustrate the support provided by the editor for automatically checking integrity constraints and deriving information in models. Integrity constraints are inspected via two different mechanisms named *live verification* and *batch verification*. In order to illustrate these features, let us make use of a simple domain model of a person, which can play the role of a student, depending on a context. This simple universe of discourse is comprised of concepts such as Person, Organization, Enrollment, Person's subtypes (*viz.* Man and Woman), Person's roles (*e.g.*, Student), Person's phases (*viz.* LivingPerson, DeceasedPerson, Child, Teenager and Adult), Person's parts (*e.g.*, Brain and Heart), Organization's roles (*e.g.*, School), Organization's phases (*e.g.* ActiveOrganization and ExtinctOrganization), Brain's phases (*viz.* FunctionalBrain and NonfunctionalBrain), Heart's phases (*viz.* FunctionalHeart and NonfunctionalHeart) and the category BiologicalOrgan.

In the following we briefly exemplify how the editor can assist the user in the construction of a simple conceptual model in this domain.

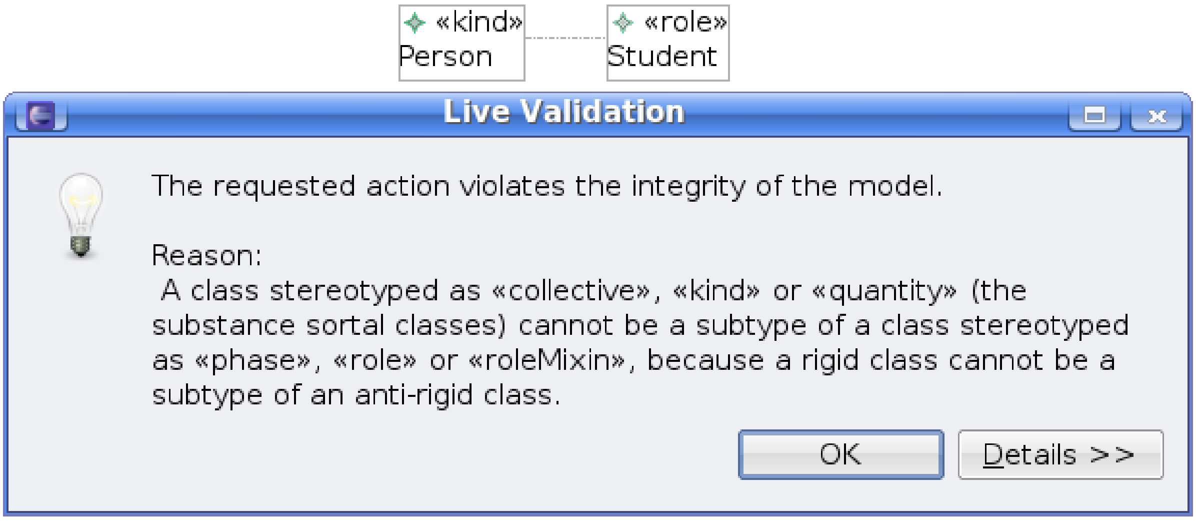
4.6.1 Live Verification

In this conceptualization, Person would typically be modeled in OntoUML as a class with a «kind» stereotype, and a Student would be modeled as a class with a «role» stereotype, as is shown in Fig. 20. In OntoUML, the «kind» stereotype is used to represent the UFO Kind category, and the «role» stereotype represents the UFO Role category.

«kind»
Person
«role»
Student

Figure 20: A simple model.

Now, as discussed in 20, a common mistake in conceptual modeling is the use of subtyping to represent alternative allowed types, *i.e.*, alternative types that supply players for a given role. In this particular case, suppose that the user attempts to represent that instances of Person are possible players of the role Student, by using subtyping. In other words, the user tries to model a Kind Person as a subtype of the Role Student. If allowed, this would not be an ontologically correct model, since it is not the case that every instance of Person is a Student, and since a Person cannot cease to be a Person but it can cease to be a Student. When attempting to create this ontologically incorrect model with the editor presented here, the integrity constraint shown in Listing 5 is violated. As consequence, the editor ignores the corresponding model updating action and prompts a live verification pop-up that alerts the user of his attempt of creating an invalid model. The verification pop-up resulting from this example is shown in Fig. 21.



The requested action violates the integrity of the model.

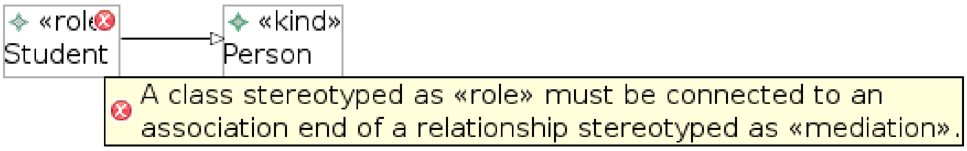
Reason:
A class stereotyped as «collective», «kind» or «quantity» (the substance sortal classes) cannot be a subtype of a class stereotyped as «phase», «role» or «roleMixin», because a rigid class cannot be a subtype of an anti-rigid class.

Figure 21: Live verification example.

Listing 5: OCL expression that constrains a rigid substantial universal to not be a subclass of an anti-rigid universal.

```
context Generalization
inv SubstanceSortalConstraint3: if self.specific.
    oclIsKindOf(SubstanceSortal) then not (self.general.
    oclIsKindOf(AntiRigidSortalClass) or self.general.
    oclIsKindOf(RoleMixin)) else true endif
```

A correct, but incomplete, modeling would make Student a subtype of Person, as shown in Fig. 22. Nevertheless, if we manually verify this model, the editor will advise that a Role must be connected to Mediation relationships, as shown in Fig. 22.



A class stereotyped as «role» must be connected to an association end of a relationship stereotyped as «mediation».

Figure 22: An incomplete solution to the model pictured in Fig 21.

The following subsection shows a correct modeling of Person and Student, by making the Role Student a subtype of the Kind Person and depicting the context in which a Person is a Student, by creating Mediation relationships and Relators.

4.6.2 Deriving Model Information

In order to represent the relation between Student and Person, one should model Student as a Role played by Person in a certain context, where he studies in a School. Analogously, one should model School as a Role played by an Organization when having Students. This context is materialized by the Material Relation study (represented as the «material» stereotype in OntoUML), which is in turn, derived from the existence of the Relator Universal Enrollment («relator»). In other words, we can say that a particular Student *x* studies in a particular School *y* iff there is an Enrollment that mediates *x* and *y*. This situation is illustrated in Fig. 23. The mediation formal relations between the Relator Enrollment and the Roles Student and School are responsible for the existence of the derived Material Relation study that holds between Student and School.

Furthermore, the upper cardinality restrictions of both extremities of the study relation can be systematically calculated from these associations as exemplified in Fig. 41 and implemented in OCL as shown in Listings 6 and 7. The derivation of study from the mediation relations is represented by a Derivation association (pictured as a dashed line association between study (that is in the source extremity) and Enrollment (that is in the target extremity, where there is a black circle)), which have its upper and lower cardinality constraints of the source extremity systematically calculated, as shown in Listings 8 and 9. Regarding the target extremity of Derivation associations, from the OntoUML profile, the upper and lower cardinality constraints must be exactly one (see the third constraint of Derivation associations, in page 49, which is formalized in OCL in Listing 47).

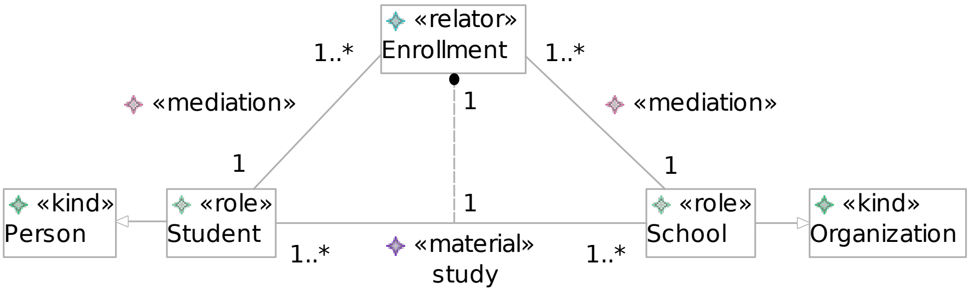


Figure 23: Example of derivation of model information.

Listing 6: OCL expression that calculates upper cardinalities for one extremity of Material Associations.

```
context MaterialAssociation::
    deriveUpperMaterialAssociationExt1():EInt
body: let der:Derivation = Derivation.allInstances()->
    select(x | x.source->any(true).oclAsType(Property).
    endType = self)->any(true), matext1:Type = self.
    associationEnd->at(1).oclAsType(Property).endType.
    oclAsType(Type), matext2:Type = self.associationEnd->at
    (2).oclAsType(Property).endType.oclAsType(Type) in (let
     rel:Relator = der.target->any(true).oclAsType(Property
    ).endType.oclAsType(Relator) in (let med1:Set(Mediation
    ) = Mediation.allInstances()->select(x | x.source->
    exists(y | y.oclAsType(Property).endType = rel) and x.
    target->exists(y | y.oclAsType(Property).endType =
    matext1)), med2:Set(Mediation) = Mediation.allInstances
    ()->select(x | x.source->exists(y | y.oclAsType(
    Property).endType = rel) and x.target->exists(y | y.
    oclAsType(Property).endType = matext2)) in (let
    med1targetupper: Integer = med1.target->any(true).
    oclAsType(Property).upper, med2sourceupper: Integer =
    med2.source->any(true).oclAsType(Property).upper in (if
     ((med2sourceupper = -1) or (med1targetupper = -1))
    then (-1) else (med2sourceupper*med1targetupper) endif)
    )))
```

Listing 7: OCL expression that calculates upper cardinalities for the other extremity of Material Associations.

```
context MaterialAssociation::
    deriveUpperMaterialAssociationExt2():EInt
body: let der:Derivation = Derivation.allInstances()->
    select(x | x.source->any(true).oclAsType(Property).
    endType = self)->any(true), matext1:Type = self.
    associationEnd->at(1).oclAsType(Property).endType.
    oclAsType(Type), matext2:Type = self.associationEnd->at
    (2).oclAsType(Property).endType.oclAsType(Type) in (let
     rel:Relator = der.target->any(true).oclAsType(Property
    ).endType.oclAsType(Relator) in (let med1:Set(Mediation
    ) = Mediation.allInstances()->select(x | x.source->
    exists(y | y.oclAsType(Property).endType = rel) and x.
    target->exists(y | y.oclAsType(Property).endType =
    matext1)), med2:Set(Mediation) = Mediation.allInstances
    ()->select(x | x.source->exists(y | y.oclAsType(
    Property).endType = rel) and x.target->exists(y | y.
    oclAsType(Property).endType = matext2)) in (let
    med1sourceupper: Integer = med1.source->any(true).
    oclAsType(Property).upper, med2targetupper: Integer =
    med2.target->any(true).oclAsType(Property).upper in (if
     ((med1sourceupper = -1) or (med2targetupper = -1))
    then (-1) else (med1sourceupper*med2targetupper) endif)
    )))
```

Listing 8: OCL expression that calculates lower cardinalities for Derivations.

```
context Derivation::deriveLowerDerivation():EInt
body: let mat:MaterialAssociation = self.source->any(true)
    .oclAsType(Property).endType.oclAsType(
    MaterialAssociation), rel:Relator = self.target->any(
    true).oclAsType(Property).endType.oclAsType(Relator) in
     (let matext1:Type = mat.associationEnd->at(1).
    oclAsType(Property).endType.oclAsType(Type), matext2:
    Type = mat.associationEnd->at(2).oclAsType(Property).
    endType.oclAsType(Type) in (let med1:Set(Mediation) =
    Mediation.allInstances()->select(x | x.source->exists(y
     | y.oclAsType(Property).endType = rel) and x.target->
    exists(y | y.oclAsType(Property).endType = matext1)),
    med2:Set(Mediation) = Mediation.allInstances()->select(
    x | x.source->exists(y | y.oclAsType(Property).endType
    = rel) and x.target->exists(y | y.oclAsType(Property).
    endType = matext2)) in (let med1targetlower: Integer =
    med1.target->any(true).oclAsType(Property).lower,
    med2targetlower: Integer = med2.target->any(true).
    oclAsType(Property).lower in (if ((med1targetlower =
    -1) or (med2targetlower = -1)) then (-1) else (
    med1targetlower*med2targetlower) endif))))
```

Listing 9: OCL expression that calculates upper cardinalities for Derivations.

```
context Derivation::deriveUpperDerivation():EInt
body: let mat:MaterialAssociation = self.source->any(true)
```

```
.oclAsType(Property).endType.oclAsType(
MaterialAssociation), rel:Relator = self.target->any(
true).oclAsType(Property).endType.oclAsType(Relator) in
 (let matext1:Type = mat.associationEnd->at(1).
oclAsType(Property).endType.oclAsType(Type), matext2:
Type = mat.associationEnd->at(2).oclAsType(Property).
endType.oclAsType(Type) in (let med1:Set(Mediation) =
Mediation.allInstances()->select(x | x.source->exists(y
 | y.oclAsType(Property).endType = rel) and x.target->
exists(y | y.oclAsType(Property).endType = matext1)),
med2:Set(Mediation) = Mediation.allInstances()->select(
x | x.source->exists(y | y.oclAsType(Property).endType
= rel) and x.target->exists(y | y.oclAsType(Property).
endType = matext2)) in (let med1targetupper: Integer =
med1.target->any(true).oclAsType(Property).upper,
med2targetupper: Integer = med2.target->any(true).
oclAsType(Property).upper in (if ((med1targetupper =
-1) or (med2targetupper = -1)) then (-1) else (
med1targetupper*med2targetupper) endif))))
```

4.6.3 Batch Verification

A more complete version of a model in this domain is shown in Fig. 24, which represents some of the parts that compose a Person. In this figure, it is represented that a Person is composed of one non-shareable (see Definition 20) essential Brain, where the “essential” tag in this part-whole relation means that the whole is existentially dependent of the part (see Definition 24). However, part-whole relations must obey the so-called weak supplementation axiom (the first syntactical constraint for the metaclass Meronymic, shown in page 55 and formalized in OCL in Listing 54), which, in simple words, states that in order to be a whole, an entity must have at least two disjoint parts. Therefore, to satisfy this axiom, if a Person is composed of one and only one Brain, it must also have another Person component as a part. Now, differently from the Person-Student subtyping example discussed in subsection 4.6.1, the lack of a second part represented in the model that would meet the requirement posed by the weak supplementation axiom can be due to a momentary incompleteness of the model. In other words, after the part-whole relation between Person and Brain is represented, the user can still include information in the model that will prevent this model from being considered ontologically inconsistent. As this example shows, there are verification actions that should only be performed by the tool once the user deems suitable. Now, as illustrated in Fig. 25, if this model is verified with the presented information, the editor prompts to the user that, in that form, the model is considered incorrect. Furthermore, the editor informs the user by highlighting the source and reason of inconsistency in the model.

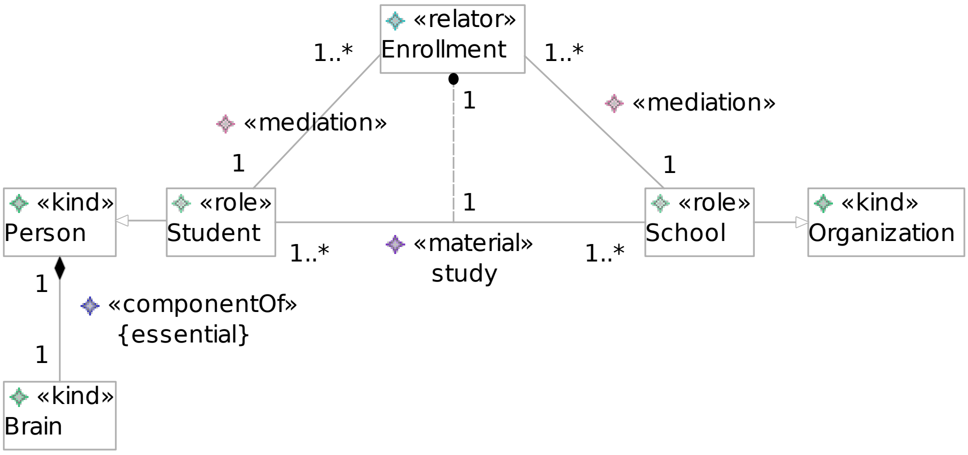
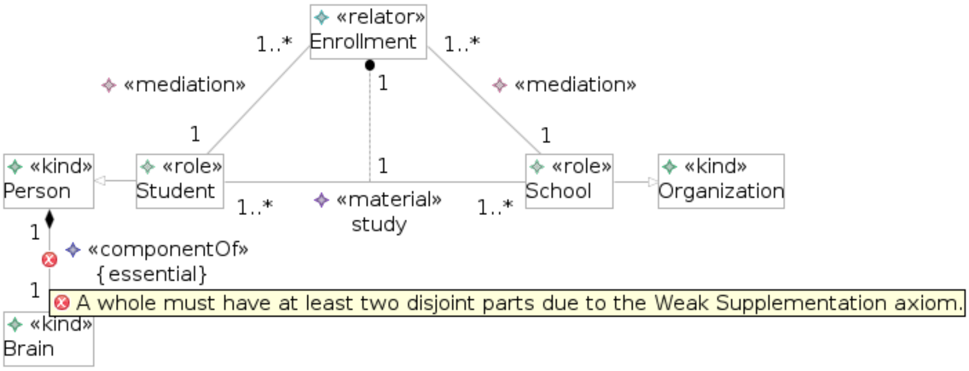


Figure 24: Batch verification example.



A whole must have at least two disjoint parts due to the Weak Supplementation axiom.

Figure 25: Verification of the OntoUML model depicted in Fig 24.

A possible solution to this issue is to represent that a Person is composed of something more than a Brain, *e.g.*, a Heart. Fig. 26 depicts this alternative representation, where a Person is composed of a non-shareable essential unique Brain and a non-shareable unique Heart.

4.6.4 A Larger Model

A more complete case study in this domain could represent that a Person is partitioned into the SubKinds Man and Woman, and is also partitioned in a Living phase and a Deceased phase. The Living phase, in its turn, is partitioned into a Child phase, a Teenager phase and an Adult one. Also, Biological Organs like Brain and Heart are partitioned into a Functional phase and a

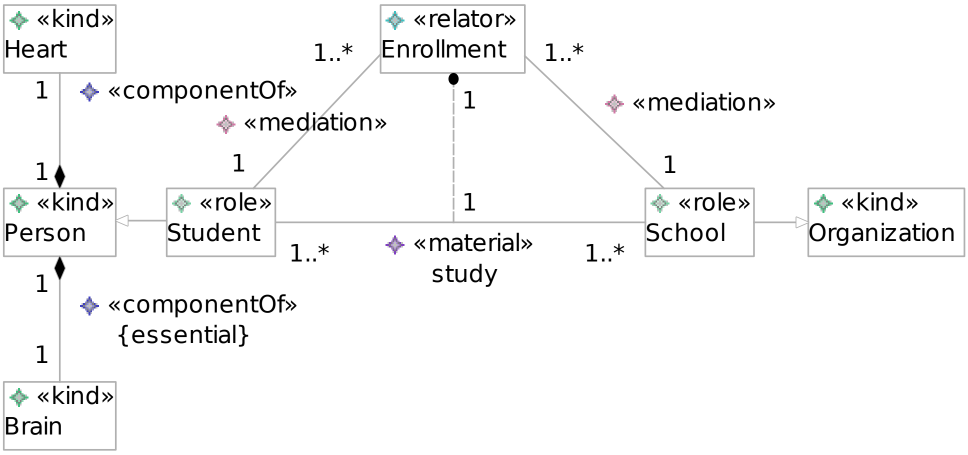


Figure 26: A possible solution to correct the model pictured in Fig. 24.

Non-functional one, which are the Phases FunctionalBrain and NonfunctionalBrain (regarding the Kind Brain), and FunctionalHeart and NonfunctionalHeart (regarding the Kind Heart).

Furthermore, when living, a Person must have an unique non-shareable and immutable FunctionalBrain as part (*i.e.*, the person cannot change its brain while living) and a FunctionalBrain must be a part of an immutable LivingPerson (*i.e.*, a brain cannot change its owner while being functional). However, a LivingPerson must have an unique FunctionalHeart as part, but he/she may change his/her heart; while a FunctionalHeart optionally may be a part of a LivingPerson. There is also a Kind Organization that is partitioned into the Phases ActiveOrganization and ExtinctOrganization. Finally, when a LivingPerson studies in an ActiveOrganization by means of an Enrollment (conversely, when an ActiveOrganization provides educational services to a LivingPerson by means of an Enrollment), then this person is said to play the role of a Student and the organization plays the role of a School. This model is depicted in Fig. 27.

It is worth to notice that, as Phases are always defined in a partition set (11, p. 103, *formulæ* 9, 10) (cited in page 33), then this model must have a covering and disjoint GeneralizationSet from all the Phases to its supertypes. Contrariwise, OntoUML does not requires GeneralizationSets for the SubKinds Man and Woman.

4.7 Conclusions

In this chapter, we explain how we build a graphical editor for building and verifying OntoUML models, by using the Eclipse plug-ins EMF, MDT and GMF. We model the OntoUML metamodel in Ecore, its syntactical constrains into OCL expressions and its concrete syntax is modeled by

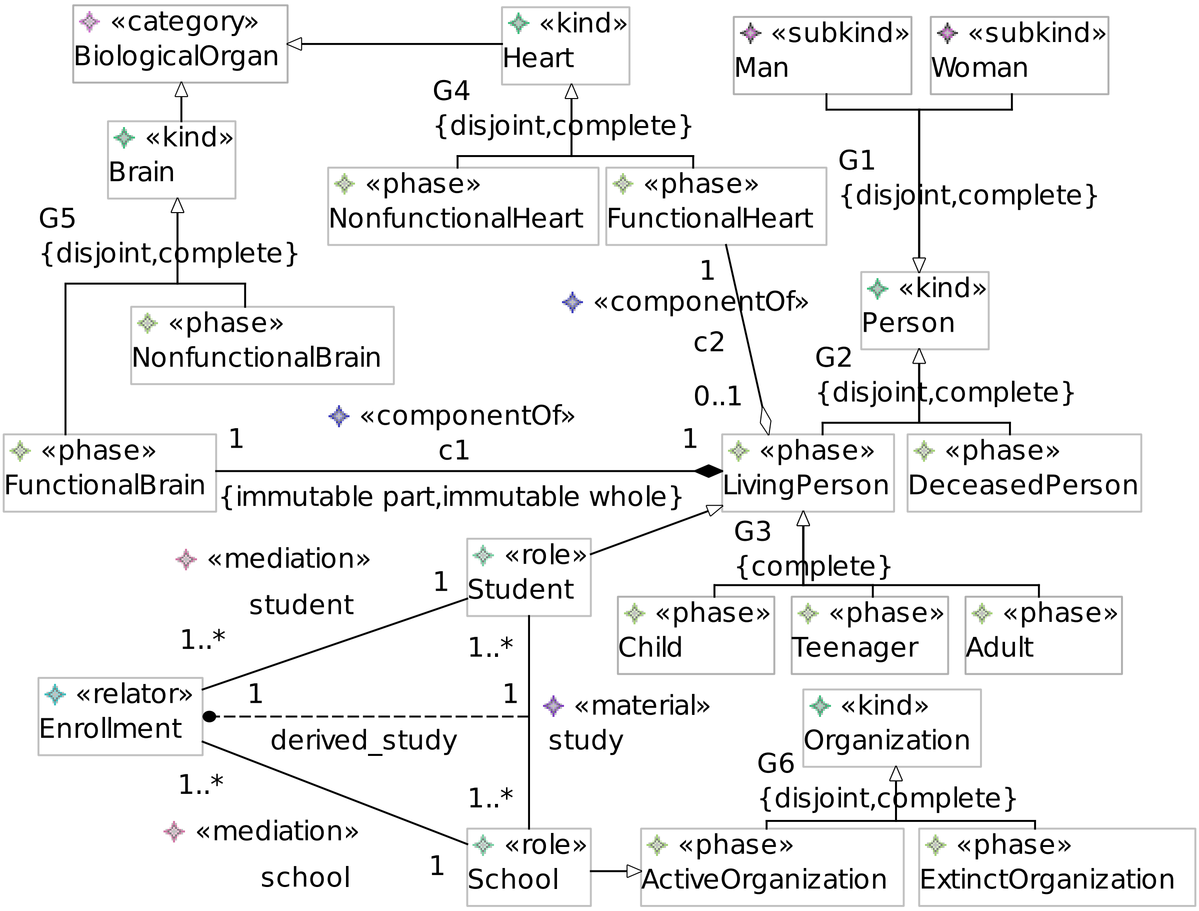


Figure 27: A larger model.

using GMF.

Also, we create new invariants and formalize them in OCL in order to extend the original set of OntoUML syntactical constraints that is posed in the OntoUML profile shown in 11, pp. 317–320, 334–338, 348–352.

We present a case study by exemplifying the two types of syntactical verification, namely Batch and Live Validation, and showing how they can assist the modeler in order to build syntactically correct OntoUML models. We also demonstrate the capability of our editor in automatically deriving and modifying the value of a number of meta-attributes, in order to save the user from specifying these values and also guaranteeing that they are consistent.

Moreover, in this chapter we implement a number of artifacts from the OntoUML specification in a declarative mode, in terms of OCL expressions. For example, we implement (i) all the OntoUML syntactical constraints, (ii) all the derived meta-attributes, and (iii) some automatic derivation of information, as OCL expressions which are handled by the MDT Eclipse plug-in (65). By implementing these artifacts in a declarative mode, we have a number of advantages when compared to a procedural codification, such as: (i) direct comparability between our implementation and the abstract constraints from the original OntoUML profile, because declarative expressions describes *what* computation should be performed instead of *how* to compute it; (ii) reusability of the metamodel through different implementations, *etc.*.

The binaries and source code of this editor are available at 59 and licensed under GPLv3 and EPLv1. Also, appendix C deals with its installation and the creation of OntoUML models.

5 A Tool for Supporting the Validation of OntoUML Models

We believe that, in general, performing validation of OntoUML models is not an easy task. Many of the ontological meta-properties incorporated into OntoUML are modal in nature and it may be difficult for human beings to reason upon the several possible changes in the instances in a set of worlds.

Therefore, in this chapter, we show how to help one to validate OntoUML models by instance generation and analysis (section 5.1). We argue that it is possible to generate instances of OntoUML models by transforming them to Alloy specifications and using the Alloy Analyzer tool to generate the instances (sections 5.2 and 5.5). Based on this premise, we utilize our QS5 formalization of the OntoUML constructs (section 5.3) in order to model the modality required by OntoUML models in Alloy by means of Kripke structures (section 5.4) and automatically transform OntoUML models to Alloy specifications by creating and using an ATLAS Transformation Language[1] (ATL) transformation from OntoUML models to Alloy specifications (appendix B). Furthermore, in section 5.6 we present the illustration of the mapping of an OntoUML model to an Alloy specification using the running example employed throughout this thesis (Fig. 27, created in section 4.6). Moreover, this section exemplifies an instance generated by the Alloy Analyzer. Section 5.6 also discusses the customization of visualization themes in the Alloy Analyzer in order to provide visualization mechanisms to the generated instances that are more amenable to human users. Finally, in section 5.7 we pose our conclusions for this chapter.

5.1 Validating OntoUML Models by Instances Analysis

In section 1.3, we said that we aim at supporting model validation by generating model instances. We believe that the analysis of a well-chosen set of these instances (*e.g.*, instances that exhibit important behaviour by dynamic classification) can improve the user's confidence in the validity of the model.

[1] http://www.eclipse.org/m2m/atl.

For example, consider that a modeler built the OntoUML model shown in Fig. 28. This model is automatically verified by the OntoUML Editor, so it is syntactically correct and seems to be a reasonable model.

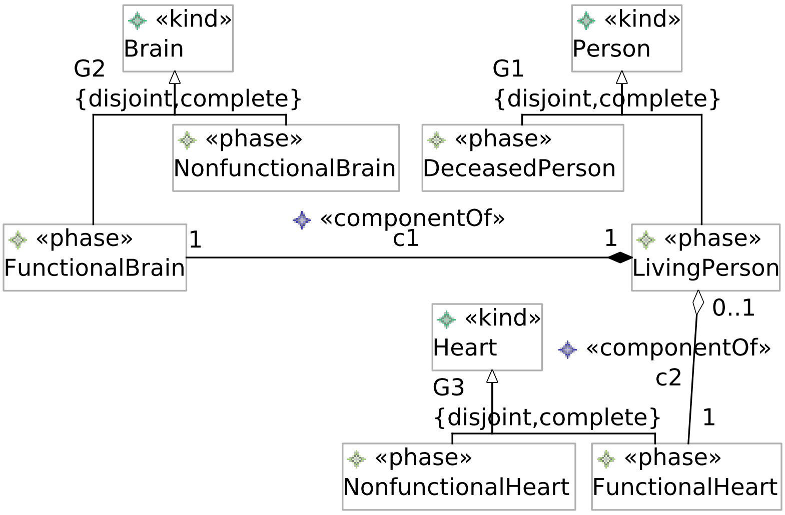


Figure 28: A verified but invalid OntoUML model.

However, this model would allow an instance in which a person changes his/her (functional) brain while living. This instance is shown in Figs. 29 and 30. Fig. 29 shows the ordering of the temporal moments, while Fig. 30 shows the dynamic classification of the atoms within the moments depicted in Fig. 29. In Fig. 29, the counterfactual world represents a world containing state-of-affairs that could have happened, but, accidentally, did not happened. In other words, counterfactuals are state-of-affairs that were possible to happen in some moment in the past, but did not happened. This instance shows that a LivingPerson “Person” has a “Brain1” in the past moment and a “Brain0” in the current moment, when he/she is still a LivingPerson.

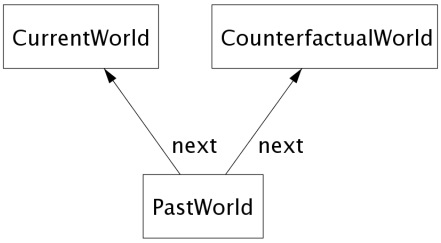
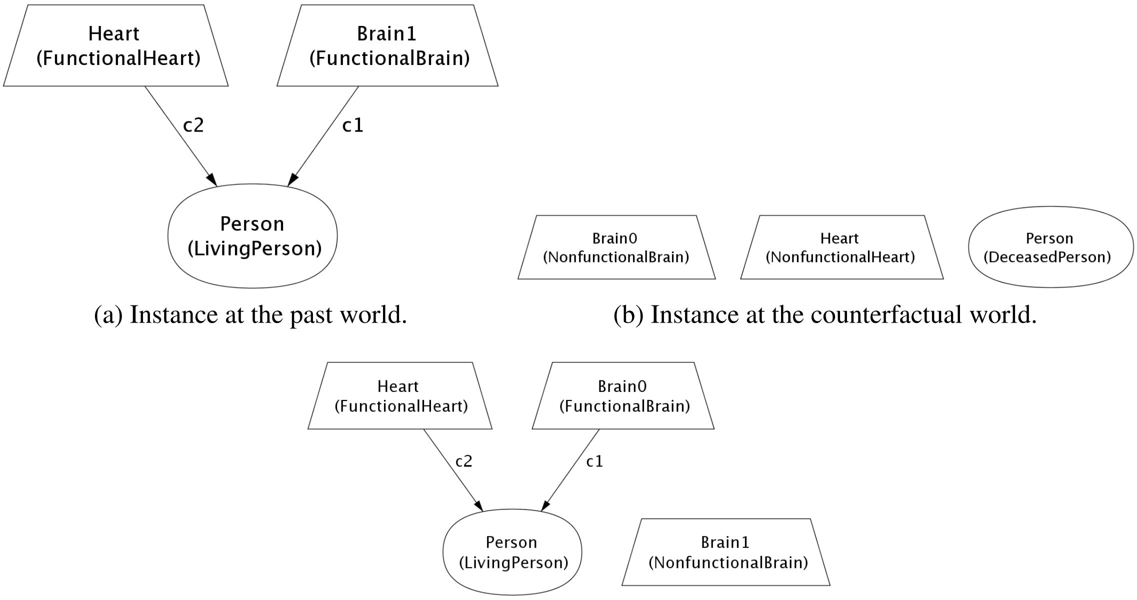


Figure 29: The temporal ordering of worlds of the instance shown in Fig. 30.

Currently, it is not feasible that a living person changes his/her brain while living. This unintended behaviour could have been detected if the modeler had this instance at hand when building this model. A possible correction is to make the «componentOf» relationship between FunctionalBrain and LivingPerson an immutable part one, not allowing a person to change his/her brain while living, as shown in Fig. 31



(a) Instance at the past world.

(b) Instance at the counterfactual world.

(c) Instance at the current world.

Figure 30: Dynamic classification of the atoms within the moments depicted in Fig. 29.

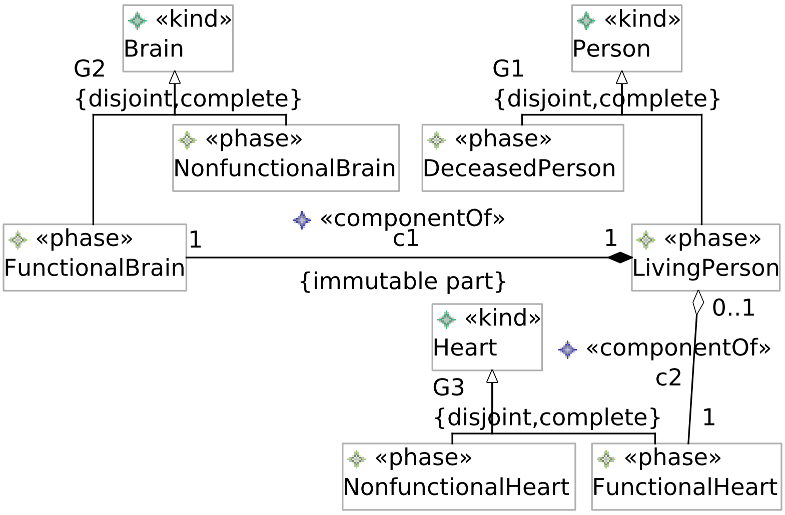
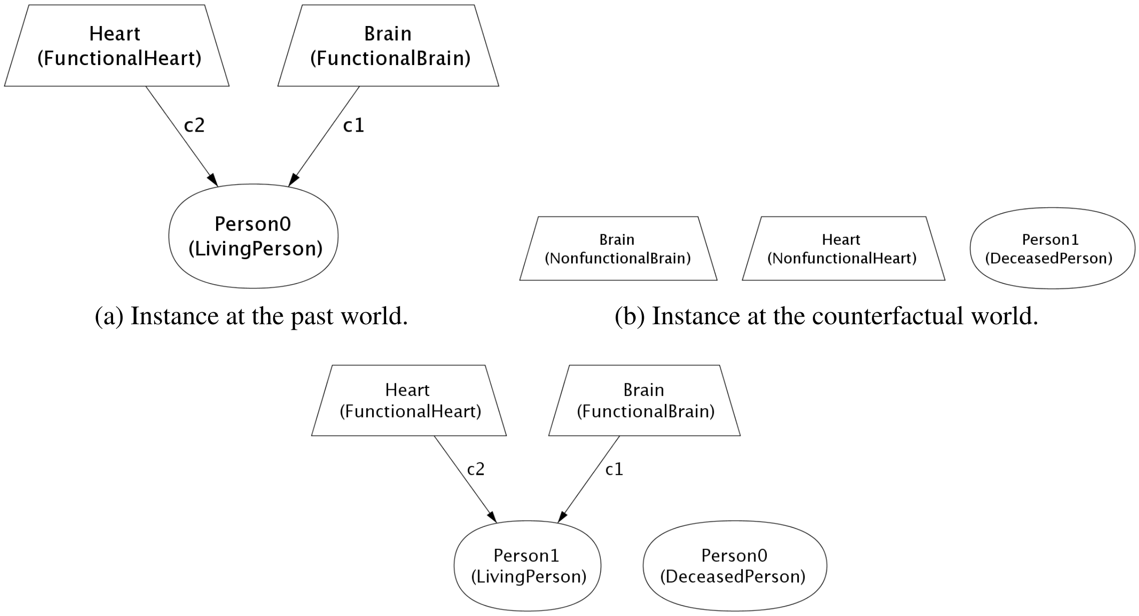


Figure 31: An attemption to correct the model shown in Fig. 28.

Nevertheless, the corrected version shown in Fig. 31 would allow a functional brain to change from a (living) person to another (living) person. This instance is shown in Figs. 29 and 32. Fig. 29 shows the ordering of the temporal moments, while Fig. 32 shows the dynamic classification of the atoms within the moments depicted in Fig. 29. This instance shows that a FunctionalBrain “Brain” is owned by “Person0” in the past moment, while being owned by a “Person1” in the current moment, when it is still a FunctionalBrain.

It is feasible that a brain go from a person to a container with formaldehyde, but, currently, while not being a part of a living person, the brain must be non-functional. As in our model we only describe meronymic relationships between functional brains and alive persons, a functional



(a) Instance at the past world.

(b) Instance at the counterfactual world.

(c) Instance at the current world.

Figure 32: Dynamic classification of the atoms within the moments depicted in Fig. 29.

brain should not be allowed to change its owner while being functional. Again, this unintended behaviour could have been detected if the modeler had this instance at hand. A possible fix to this model is shown in Fig. 33, where the «componentOf» relationship between Brain and Person is also an immutable whole one, not allowing a brain to change its owner while being functional.

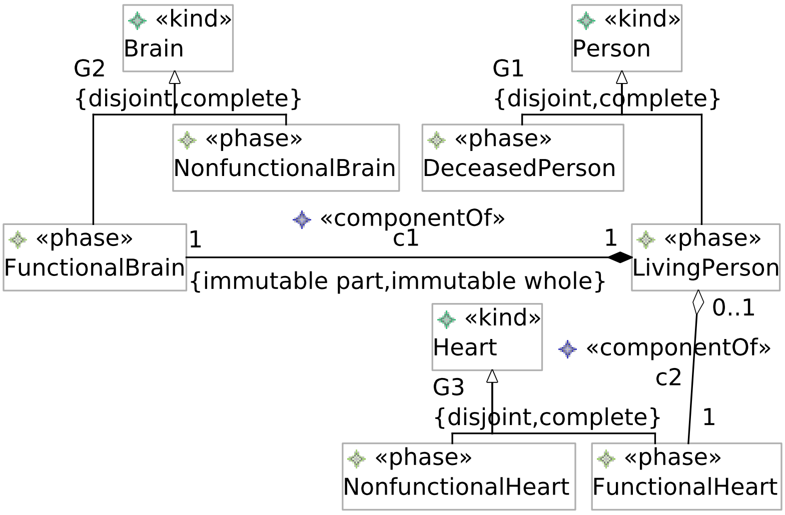


Figure 33: A possible correction to the model shown in Fig. 31.

Therefore, we consider that a tool capable of automatically generating instances of conceptual models would be valuable for the validation phase, when the dynamic classification of instances would exhibit unintended behaviours or improve modelers' confidence in the validity of the model.

Thus, in order to automatically generate instances of OntoUML models, we built a model transformation from OntoUML to Alloy[2], respecting UFO's (modal) axioms, so we can use the Alloy Analyzer to automatically generate instances.

5.2 Architecture

The architecture of this model transformation is shown in Fig. 34. From an OntoUML model, we apply an OntoUML to Alloy model transformation (described in section 5.5) and get an Alloy specification. Within the Alloy Analyzer, we can automatically generate instances for this specification, which are composed of *atoms* and relations between *atoms*.

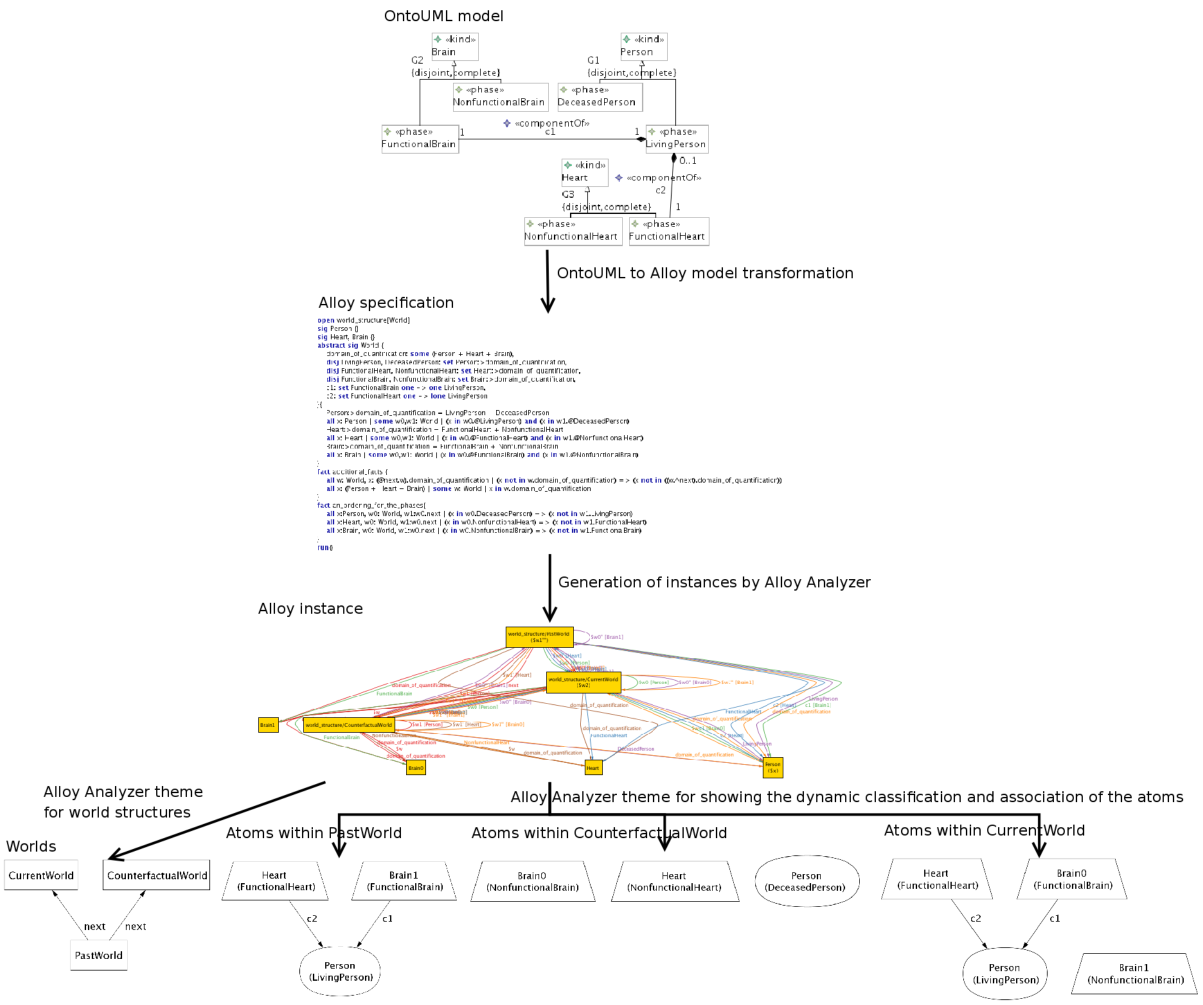
In order to facilitate the validation of the OntoUML models, we want the generated atoms and relations to be ordered in a branching time temporal sequence, so the modeler can perceive the dynamic classification of the atoms. However, as Alloy has no built in notion of temporal sequence, we create (i) a set of atoms to represent temporal worlds; and (ii) a partial order relation from atoms representing temporal worlds to atoms representing OntoUML constructs, indicating which atoms exist (*i.e.*, are in the domain of quantification) of each temporal world. Therefore, if we apply our OntoUML to Alloy model transformation to an OntoUML model M, and subsequently use the Alloy Analyzer to generate instances of M, then for each generated instance, the atoms that are not temporal worlds are interpreted as momentary states of instances of the OntoUML classifiers taken from M. Section 5.4 shows how we model these branching time temporal worlds in Alloy.

Moreover, when the Alloy Analyzer generates instances for OntoUML models, the temporal worlds and existence relations are represented just as any other Alloy atom/relation, leading to instances that are not suitable to be inspected and reasoned upon by the human modeler. However, the Alloy Analyzer allows the creation of visualization themes (66) for customizing the visualization of the generated atoms and relations.

Therefore, we create two visualization themes, one for the visualization of the branching time temporal ordering of the worlds (shown in Listing 117 in appendix E), and the other for the visualization of temporal snapshots by projecting the generated atoms and relations in each world, so the user can select a world in order to see which atoms and relations exist in that world and how they are classified (shown in Listing 118 in appendix E).

Finally, the Alloy Analyzer tool can only search for instances within a restricted context,

[2] This transformation is specified in ATL and is capable of automatically generating Alloy specifications from OntoUML models. The complete transformation is shown in appendix B.



```
open world_structure[World]
sig Person {}
sig Heart, Brain {}
abstract sig World {
    domain_of_quantification: some (Person + Heart + Brain),
    disj LivingPerson, DeceasedPerson: set Person:>domain_of_quantification,
    disj FunctionalHeart, NonfunctionalHeart: set Heart:>domain_of_quantification,
    disj FunctionalBrain, NonfunctionalBrain: set Brain:>domain_of_quantification,
    c1: set FunctionalBrain one -> one LivingPerson,
    c2: set FunctionalHeart one -> lone LivingPerson
}{
    Person:>domain_of_quantification = LivingPerson + DeceasedPerson
    all x: Person | some w0,w1: World | (x in w0.@LivingPerson) and (x in w1.@DeceasedPerson)
    Heart:>domain_of_quantification = FunctionalHeart + NonfunctionalHeart
    all x: Heart | some w0,w1: World | (x in w0.@FunctionalHeart) and (x in w1.@NonfunctionalHeart)
    Brain:>domain_of_quantification = FunctionalBrain + NonfunctionalBrain
    all x: Brain | some w0,w1: World | (x in w0.@FunctionalBrain) and (x in w1.@NonfunctionalBrain)
}
fact additional_facts {
    all w: World, x: (@next.w).domain_of_quantification | (x not in w.domain_of_quantification) => (x not in ((w.^next).domain_of_quantification))
    all x: (Person + Heart + Brain) | some w: World | x in w.domain_of_quantification
}
fact an_ordering_for_the_phases{
    all x:Person, w0: World, w1:w0.next | (x in w0.DeceasedPerson) => (x not in w1.LivingPerson)
    all x:Heart, w0: World, w1:w0.next | (x in w0.NonfunctionalHeart) => (x not in w1.FunctionalHeart)
    all x:Brain, w0: World, w1:w0.next | (x in w0.NonfunctionalBrain) => (x not in w1.FunctionalBrain)
}
run{}
```

Generation of instances by Alloy Analyzer
Alloy instance
world_structure/PastWorld ($w1'')
world_structure/CurrentWorld ($w2)
world_structure/CounterfactualWorld
Brain1
Brain0
Heart
Person ($s1)
domain_of_quantification
Alloy Analyzer theme for world structures
Alloy Analyzer theme for showing the dynamic classification and association of the atoms
Worlds
CurrentWorld
CounterfactualWorld
next
next
PastWorld
Atoms within PastWorld
Heart (FunctionalHeart)
Brain1 (FunctionalBrain)
c2
c1
Person (LivingPerson)
Atoms within CounterfactualWorld
Brain0 (NonfunctionalBrain)
Heart (NonfunctionalHeart)
Person (DeceasedPerson)
Atoms within CurrentWorld
Heart (FunctionalHeart)
Brain0 (FunctionalBrain)
c2
c1
Person (LivingPerson)
Brain1 (NonfunctionalBrain)

Figure 34: Architecture.

i.e., a given finite maximum number of *atoms*. So, unlike theorem provers, Alloy Analyzer can not prove that a specification is unsatisfiable. But, if Alloy Analyzer finds an instance, then the Alloy specification is satisfiable (semantically consistent)[3]. If the tool can not find an instance, either the Alloy specification is unsatisfiable or the given context was too small. This is why Alloy is considered a "lightweight formal method" (Bowen *et al.*(67), 1996 apud Jackson(25), 2006, p. xiii).

5.3 Modeling OntoUML Constructs in QS5

In this section, in order to facilitate the understanding of the dynamics of creation and destruction of OntoUML instances within worlds, we revisit the possibilist formalization of some OntoUML

[3] We can only guarantee that an OntoUML model is satisfiable given the satisfiability of the Alloy specification if we can guarantee the formal correctness of the transformation between models of the former kind to the latter. In this thesis, due to time limitations, this proof could not be constructed. As a consequence, we assume here the correctness of this transformation.

and UFO concepts (11, 55) in the QS5 actualist (*i.e.*, having a varying domain of quantification) quantified modal logic. We also make some minor modifications in a number of definitions that will be pointed by specific remarks in this section.

By revisiting the possibilist formalization into an actualist one, the existence predicate ε will be identified with the pertinence of individuals within the varying domain of quantification $\mathcal{D}(w)$, as shown in section 2.1. Therefore, in an actualist formalization, there is no need to define an explicit existence predicate ε, which would be necessary in a possibilist approach, in case there is a need to predicate about the existence of individuals within worlds. Moreover, this alternative formalization can be reutilized by other people for different purposes.

In the following, we will model the modal definitions that we use to characterize some key notions of UFO, like rigidity, anti-rigidity, relational dependence, existential dependence, specific dependence, essential part, immutable part, generic dependence, mandatory part, inseparable part, mandatory whole and immutable whole. As shown in this section, some of the original definitions posed in 11, 55 had to be revised to comply with this new actualist mode.

In 55, p. 9, rigidity and anti-rigidity are defined in the following way:

> **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': $R(\mathrm{T}) \triangleq \Box(\forall x(\mathrm{T}(x) \rightarrow \Box(\mathrm{T}(x))))$. ■(55, p. 9)
> **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: $AR(\mathrm{T}) \triangleq \Box(\forall x(\mathrm{T}(x) \rightarrow \Diamond(\neg \mathrm{T}(x))))$. ■(55, p. 9)

Those possibilist definitions are neutral regarding the existence of individuals within worlds. In our actualist revisitations of Rigidity and Anti-Rigidity, we make explicit our commitments with the existence (*i.e.*, pertinence) of individuals within the domains of quantification of the worlds.

Definition 21 (Rigidity (revisited)): A Universal U is rigid if for every instance x of U, x is necessarily (in the modal sense) an instance of U. In other words, if x instantiates U in a given world w, then x must instantiate U in every world w' in which x exists and that is accessible from w: $R(\mathrm{U}) \triangleq \Box(\forall x(\mathrm{U}(x) \rightarrow \Box(\varepsilon(x) \rightarrow \mathrm{U}(x))))$. ■

Definition 22 (Anti-Rigidity (revisited)): A Universal U is anti-rigid if for every instance x of U, x possibly (in the modal sense) exists not being an instance of U. In other words, if x instantiates U in a given world w, then there is a possible world w', accessible from w, in which x exists and in which x does not instantiate U: $AR(\mathrm{U}) \triangleq \Box(\forall x(\mathrm{U}(x) \rightarrow \Diamond(\varepsilon(x) \land \neg \mathrm{U}(x))))$. ■

We will not revisit Relational Dependence and Existential Dependence, as their formalization in 55, p. 10 and 55, p. 11, respectively, are suitable for both possibilist and actualist interpretations. However, we remark that we are assuming that every individual in the domains of quantification possibly exists and possibly do not exist (*i.e.*, there is no necessarily existing individuals) (11). Therefore, the definition of Existential Dependence is not trivially satisfied. For example, for an arbitrary individual x: (i) if x were allowed to exist in no worlds (*i.e.*, if x could necessarily not exist), then for every individual y, *ed*(x,y) would be trivially true; and (ii) if x were allowed to exist in every world (*i.e.*, if x could necessarily exist), then for every individual y, *ed*(y,x) would be trivially true.

> **Definition (Relational Dependence)**: A type T is relationally dependent on another type P via relation R iff for every instace x of T, there is an instance y of P such that x and y are related via R: $R(\mathrm{T},\mathrm{P},\mathrm{R}) \triangleq \Box(\forall x(\mathrm{T}(x) \rightarrow \exists y(\mathrm{P}(y) \wedge \mathrm{R}(x,y))))$. ■ (55, p. 10)
> **Definition (Existential Dependence)**: Let the predicate ε denote existence[4]. We have that an individual x is existentially dependent on another individual y iff, as a matter of necessity, y must exist whenever x exists, or formally: $ed(x,y) \triangleq \Box(\varepsilon(x) \rightarrow \varepsilon(y))$. ■ (55, p. 11)

Moreover, we will change some statements from section 3.3. We will distinguish the two types of part-whole relations based on the distinction between *specific dependence* (Definition 23) and the one of generic dependence (Definition 26) instead of *existential dependence* (Definition 1) and the one of generic dependence (Definition 14).

Definition 23 (specific dependence): For every individuals x and y, and Universals U_1 and U_2: $SD(x,y,\mathrm{U}_1,\mathrm{U}_2) \triangleq \Diamond(\varepsilon(x) \wedge \mathrm{U}_1(x)) \wedge \Box((\varepsilon(x) \wedge \mathrm{U}_1(x)) \rightarrow (\varepsilon(y) \wedge \mathrm{U}_2(y)))$. In other words, whenever x exists instantiating U_1, then y must exist instantiating U_2. ■

As one can observe by contrasting the definitions 23 and 26, the former is a relation between two individuals, whilst the latter is a relation between an individual and a universal.

Let us explain the reason we substitute existential dependence by specific dependence by means of the model shown in Fig. 27. In this model, the parthood relationship c1 between FunctionalBrain and LivingPerson does not imply existential dependence, because when a LivingPerson *John* ceases to be a LivingPerson, he may continue existing (being a DeceasedPerson) but it is possible (in the modal sense) that his previous Brain b no longer exists. The relationship c1 is one of specific dependence, because it is necessary (in the modal sense) that the Brain b exists (being a FunctionalBrain) only when *John* is a LivingPerson. Therefore, unlike essential or inseparable parthood relations, immutable ones do not imply existential dependence. However, essential, inseparable or immutable parthood relations imply specific dependence.

[4] See footnote 2 of chapter 3 for the predicate ε.

Contrariwise, a mandatory parthood relation is one that implies generic dependence from the part to the whole (mandatory whole, Definition 29) or from the whole to the part (mandatory part, Definition 27).

We can then further qualify the division of the parthood relations regarding modality, which was presented in section 3.3:

i The relations that imply specific dependence (Definition 23), such as relations that are essential (Definition 24), inseparable (Definition 28), immutable regarding the part (Definition 25) or immutable regarding the whole (Definition 30);

ii The relations that imply generic dependence (Definition 26), such as relations that are mandatory regarding the part (Definition 27) or mandatory regarding the whole (Definition 29)

Furthermore, in 55, p. 11, essential parthood is defined in the following way[5]:

> **Definition (essential part)**: An individual x is an essential part of another individual y iff, y is existentially dependent on x and x is, necessarily, a part of y: $EP(x,y) \triangleq ed(y,x) \land \Box(x \leq y)$. This is equivalent to stating that $EP(x,y) \triangleq \Box((\varepsilon(y) \rightarrow \varepsilon(x)) \land (x \leq y))$. We adopt here the *mereological continuism* defended by 58, which states that the part-whole relation should only be considered to hold among existents, *i.e.*, $\forall x,y((x \leq y) \rightarrow (\varepsilon(x) \land \varepsilon(y)))$. As a consequence, we can have this definition in its final simplification: $EP(x,y) \triangleq \Box(\varepsilon(y) \rightarrow (x \leq y))$. ■(55, p. 11)

In the definition of essential part given above, the last formula is not a simplification of the first *formulæ*, as it determines that x must only exist when y exists and not that x and y must exist in every world. We also have to define an actualist formula for the *mereological continuism*, because the possibilist one given above is a tautology in an actualist interpretation. Moreover, in order to make a distinction between the *de re* and *de dicto* modalities[6], 55, p. 16 suggests to use the term "essential part" only when the whole is rigid. Therefore, we will define essential parthood as follows: **Definition 24 (essential part (revisited))**: For every individuals x and y, and Universals U_1 and U_2: $EP(x,y,\mathrm{U}_1,\mathrm{U}_2) \triangleq R(\mathrm{U}_2) \land \Diamond(\varepsilon(y) \land \mathrm{U}_2(y)) \land \Box((\varepsilon(y) \land \mathrm{U}_2(y)) \rightarrow (\varepsilon(x) \land \mathrm{U}_1(x) \land (x \leq y)))$. Moreover, we adopt here the *mereological continuism* defended by Simons (apud Guizzardi(55), 2007, p. 11), which states that the part-whole relation should only be considered to hold among existents, *i.e.*, $\forall x,y\ \Box((x \leq y) \rightarrow (\varepsilon(x) \land \varepsilon(y)))$. As a consequence of the *mereological continuism*, of the rigidity of U_2 and of the possibility of y to exist instantiating U_2,

[5] Following Simons (apud Guizzardi(55), 2007, p. 11) we use the symbols $\leq$ and $<$ to represent parthood and proper parthood, respectively, and we have that $(x \leq y) \triangleq (x < y) \lor (x = y)$.

[6] For *de re* and *de dicto* modalities, see 55.

we have the following simplification: $EP(x,y,\mathrm{U}_1,\mathrm{U}_2) \triangleq R(\mathrm{U}_2) \wedge \Diamond(\varepsilon(y) \wedge \mathrm{U}_2(y)) \wedge \Box(\varepsilon(y) \rightarrow (\mathrm{U}_1(x) \wedge (x \leq y)))$, *i.e.*, U_2 is rigid and it is possible that y exists instantiating U_2 and it is necessary that whenever y exists, then x must instantiate U_1 and be part of y. ■

Also, 11, p. 286 and 55, p. 17 suggest that when the whole is anti-rigid, we shall call the part immutable instead of essential. As every essential part is also immutable (11, p. 286)(55, pp. 17-18), then essential part is a specialization of immutable part. While immutable parts hold for rigid, semi-rigid or anti-rigid wholes, essential parts are maintained only between a part and a rigid whole. Therefore, we will define immutable part in the following way: **Definition 25 (immutable part)**: For every individuals x and y, and Universals U_1 and U_2: $IP(x,y,\mathrm{U}_1,\mathrm{U}_2) \triangleq \Diamond(\varepsilon(y) \wedge \mathrm{U}_2(y)) \wedge \Box((\varepsilon(y) \wedge \mathrm{U}_2(y)) \rightarrow (\mathrm{U}_1(x) \wedge (x \leq y)))$, *i.e.*, it is necessary that whenever y exists instantiating U_2, then x must instantiate U_1 and be a part of y. ■

In 55, p. 12, Guizzardi defines generic dependence and mandatory part in the following manner:

> **Definition (generic dependence)**: An individual y is generically dependent on a type T iff, whenever y exists it is necessary that an instance of T exists. This can be formally characterized by the following *formula schema*: $GD(y,\mathrm{T}) \triangleq \Box(\varepsilon(y) \rightarrow \exists \mathrm{T}, x(\varepsilon(x)))$. ■ (55, p. 12)
> **Definition (mandatory part)**: An individual x is a mandatory part of another individual y iff, y is generically dependent of a type T that x instantiates, and y has, necessarily, as a part an instance of T: $MP(\mathrm{T},y) \triangleq \Box(\varepsilon(y) \rightarrow \exists \mathrm{T}, x(x < y))$. ■ (55, p. 12)

However, we want our definition of generic dependence to be more generic, constraining individuals regarding the universals they instantiate instead of their existence. In other words, we will revisit generic dependence in a *de dicto* interpretation instead of the actual *de re* one: **Definition 26 (generic dependence (revisited))**: For every individual x and Universals U_1 and U_2: $GD(x,\mathrm{U}_1,\mathrm{U}_2) \triangleq \Diamond(\varepsilon(x) \wedge \mathrm{U}_1(x)) \wedge \Box((\varepsilon(x) \wedge \mathrm{U}_1(x)) \rightarrow \exists y\ \mathrm{U}_2(y))$. In other words, whenever x exists instantiating U_1 it is necessary that an instance of U_2 exists. ■

Regarding mandatory parts, we will also apply some results of 55, p. 17 regarding *de re* and *de dicto* modalities. In a nutshell, consider the OntoUML model pictured in Fig. 27, where a Person may be a LivingPerson and every LivingPerson must have at least one FunctionalHeart. The definition of mandatory part given above will not hold in an instance containing a world w in which a person is living and have one functional heart, and a world $w^{\backprime}$ in which he/she is deceased and have no heart. This problem arises from the fact that, actually, a person is constrained to have at least one functional heart only when being alive and not in every world he/she exists, as the definition of mandatory part states. Therefore, we will redefine mandatory part: **Definition 27 (mandatory part (revisited))**: For every individual x and Universals U_1

and U$_2$: $MP(\mathrm{U}_1,\mathrm{U}_2,x) \triangleq \Diamond(\varepsilon(x) \wedge \mathrm{U}_1(x)) \wedge \Box((\varepsilon(x) \wedge \mathrm{U}_1(x)) \rightarrow \exists y(\mathrm{U}_2(y) \wedge (y < x)))$, *i.e.*, it is necessary that whenever x exists instantiating U$_1$, then there must exist an individual as a part of x and instantiating U$_2$. ■

Also, in 55, p. 14, inseparable part and mandatory whole are defined in the following way:

> **Definition (inseparable part)**: An individual x is an inseparable part of another individual y iff, x is existentially dependent on y, and x is, necessarily, a part of y: $IP(x,y) \triangleq \Box(\varepsilon(x) \rightarrow (x \leq y))$. ■ (55, p. 14)
> **Definition (mandatory whole)**: An individual y is a mandatory whole for another individual x iff, x is generically dependent on a type T that y instantiates, and x is, necessarily, part of an individual instantiating T: $MW(\mathrm{T},x) \triangleq \Box(\varepsilon(x) \rightarrow \exists \mathrm{T},y(x < y))$. ■ (55, p. 14)

For the same reasons that we redefined essential part, we will redefine inseparable part, and for the same reasons that we redefined mandatory part, we will redefine mandatory whole.

Definition 28 (inseparable part (revisited)): For every individuals x and y, and Universals U$_1$ and U$_2$: $IP(x,y,\mathrm{U}_1,\mathrm{U}_2) \triangleq R(\mathrm{U}_2) \wedge \Diamond(\varepsilon(x) \wedge \mathrm{U}_2(x)) \wedge \Box(\varepsilon(x) \rightarrow (\mathrm{U}_1(y) \wedge (x \leq y)))$,*i.e.*, U$_2$ is rigid and it is possible that x exists instantiating U$_2$ and it is necessary that whenever x exists, then y must instantiate U$_1$ and be a whole for x. ■

Definition 29 (mandatory whole (revisited)): For every individual x and Universals U$_1$ and U$_2$: $MW(\mathrm{U}_1,\mathrm{U}_2,x) \triangleq \Diamond(\varepsilon(x) \wedge \mathrm{U}_1(x)) \wedge \Box((\varepsilon(x) \wedge \mathrm{U}_1(x)) \rightarrow \exists y(\mathrm{U}_2(y) \wedge (x < y)))$, *i.e.*, it is necessary that whenever x exists instantiating U$_1$, then there must exist an individual as a whole for x and instantiating U$_2$. ■

Although Guizzardi, in 55, does not generalize inseparable part into a new concept that holds between a non-rigid part and a whole, we create such a distinction in the OntoUML metamodel by creating the new meta-attribute “isImmutableWhole” in the metaclass Meronymic, as shown in Fig. 19. Therefore, we will define immutable whole in the following way: **Definition 30 (immutable whole)**: For every individuals x and y, and Universals U$_1$ and U$_2$: $IW(x,y,\mathrm{U}_1,\mathrm{U}_2) \triangleq \Diamond(\varepsilon(x) \wedge \mathrm{U}_2(x)) \wedge \Box((\varepsilon(x) \wedge \mathrm{U}_2(x)) \rightarrow (\mathrm{U}_1(y) \wedge (x \leq y)))$. In other words, it is necessary that whenever x exists instantiating U$_2$, then y must instantiate U$_1$ and be a whole for x. ■

5.4 Modeling Kripke Structures in Alloy

We represent modality explicitly in the generated Alloy specification. This means that this specification reifies the notion of an actualist Kripke structure. This is necessary to specify UFO’s modal axioms, given no notion of modality is built-in in Alloy.

By assuming QS5, the modality underlying UFO's axioms do not point necessarily to a temporal interpretation. However, in order to facilitate the validation stage, we adopt here a representation of a Kripke structure in which the worlds represent common sense temporal snapshots. In our ordinary language, we are able to talk about the current moment, the past, the possible future, and the facts that could have happened, but, accidentally, did not happened (*i.e.*, the counterfactuals). Therefore, we want our worlds to be interpreted as past worlds, future ones, counterfactual ones or the current one. In order to avoid *necessitas per accidens* (accidental necessity), which refers to the "contingent necessity" of some worlds to become past worlds instead of counterfactual ones, we state that a counterfactual world w_0 was possible to happen in the *very same sense* that a past world w_1 was possible to happen, because when a certain past world w_2 (prior than w_0 and w_1) was the current world, the worlds w_0 and w_1 were future worlds in the very same sense. In other words, there is no quality of some future worlds that constrains them to become past worlds instead of counterfactual ones. Therefore, the only difference between past worlds and counterfactual ones is that the first are (accidentally) memories while the latter are (accidentally) constructions of imagination.

Furthermore, we will represent the actualist Kripke structure for QS5 in Alloy and validate UFO's axiom's in their QS5 form, but we will also categorize the worlds into four disjoint categories, *viz.* PastWorld, CurrentWorld, FutureWorld and CounterfactualWorld, ordering them by a partial order relation of immediate succession. By constraining the world structures in this way, we will avoid generating instances that cannot present this temporal interpretation. Furthermore, by taking this approach, the total accessibility relation can be suppressed in the alloy implementation, because the modal operators of possibility ($\Diamond$) and necessity ($\Box$) will always take worlds in the set of all worlds ($\mathcal{W}$).

Also, the actualist domain of quantification $\mathcal{D}(w)$ is a function mapping a world to a set of objects taken from a larger set, which could be the one of *possibilia* $\mathcal{D}$, including all possible entities (substantial individuals) independently of their actual existence.

Listing 10 shows the modeling of this temporal structure in Alloy. We had to model the different types of worlds and their respective constraints regarding what types and quantities of worlds can be accessed by their immediate succession relations (named "next" in Listing 10). Also, we impose that (i) there cannot be temporal cycles (line 5 of Listing 10); (ii) a world can be the immediate next moment of at maximum one world (line 6 of Listing 10); and (iii) every world, excepting the current one, must reach the current world (lines 13, 17 and 21 of Listing 10).

Therefore, we distinguish four pairwise disjoint types of worlds in Alloy: CurrentWorld (which is a singleton representing the current moment), PastWorld (whose instances forms a linear sequence of past moments, reaching the CurrentWorld), FutureWorld (whose instances forms a tree in which the root is the CurrentWorld) and CounterfactualWorld (whose instances forms a tree in which the root is a PastWorld).

Listing 10: Kripke structure in Alloy.

```
module world_structure[World]
some abstract sig TemporalWorld extends World{
  next: set TemporalWorld -- Immediate next moments.
}{
  this not in this.^(@next) -- There are no temporal cycles.
  lone ((@next).this) -- A world can be the immediate next moment of at maximum one world.
}
one sig CurrentWorld extends TemporalWorld {} {
  next in FutureWorld
}
sig PastWorld extends TemporalWorld {} {
  next in (PastWorld + CounterfactualWorld + CurrentWorld)
  CurrentWorld in this.^@next -- All past worlds can reach the current moment.
}
sig FutureWorld extends TemporalWorld {} {
  next in FutureWorld
  this in CurrentWorld.^@next -- All future worlds can reach the current moment.
}
sig CounterfactualWorld extends TemporalWorld {} {
  next in CounterfactualWorld
  this in PastWorld.^@next -- All counterfactual worlds can reach the current moment.
}
```

5.5 The Mapping from OntoUML Models to Alloy Specifications

In this section, we present a mapping from OntoUML to Alloy. For exemplification purposes, we will make references to the Listing 15, which is the Alloy counterpart of the OntoUML model pictured in Fig. 27.

In philosophy, sortals are entities that are responsible for providing principles of identity and individuation for its instances (11). Therefore, a rigid sortal is a sortal whose instances cannot cease to be its instances without ceasing to exist. Furthermore, SubstanceSortals (*viz.* Kinds, Collectives and Quantities) are the ultimate rigid sortals that provide the principle of identity for their instances. Therefore, in Alloy, SubstanceSortals are modeled as top-level signatures (lines 2, 4 of Listing 15). As top-level signatures, (i) the instances of these classes are automatically pairwise disjoint, what is suitable because these instances are meant to be distinct objects carrying distinct identities; and (ii) an instance of a class C is always instance of C, independently of worlds, what reifies the notion of rigidity (Definition 21).

As we are adopting an actualist domain of quantification, then for every world w there is a non-empty relation named "domain_of_quantification" (see line 14 of Listing 15) representing its (non-empty) domain of quantification $\mathcal{D}(w)$, which contains a number of (w,ts) tuples in which ts is a top-level signature.

A SubKind must be (directly or indirectly) a refinement of a Kind, and we represent it in Alloy by making SubKinds as subsignatures, and using the Alloy keyword "`in`" followed by the signature representing its supertype (Definition 4). If there is a GeneralizationSet constraining some SubKinds to be disjoint (see Definition 6), we declare them with the keyword "`extends` " instead of "`in`" (see line 3 of Listing 15); and if there is a GeneralizationSet constraining them to be complete (see Definition 5), we declare a signature fact within the signature of the supertype constraining the set of instances of the supertype to be equal to the union of the sets of its subtype instances. Finally, if there is a GeneralizationSet constraining the subtypes to partition the supertype (see Definition 7), then we can substitute the signature fact by the keyword "`abstract`" before the supertype signature (as shown in line 2 of Listing 15).

Furthermore, regarding the Classifier meta-attribute isAbstract, from 12, p. 83 and 13, p. 52 we have that:

> "If true, the Classifier does not provide a complete declaration and can typically not be instantiated. An abstract classifier is intended to be used by other classifiers (*e.g.*, as the target of general metarelationships or generalization

> relationships). Default value is false.” (12, p. 83),(13, p. 52)

Therefore, whenever an abstract classifier have subtypes, we will model it as its subtypes form a complete GeneralizationSet. If an abstract classifier C has no subtypes, then there is no need to model C in Alloy, as it will not be able to have atoms.

Just as Sortal Universals, Moment Universals also provide a principle of identity for their instances, but this principle is dependent on the principles provided by the Universals they characterize or mediate, in case of Intrinsic Moment Universals (*viz.*, Quality Universals and Mode Universals) or Relator Universals, respectively. However, despite differentiating Kinds and SubKinds, OntoUML makes no distinction between the ultimate Moment Universals and the Moment Universals that are subtypes of the former, inheriting its unique principle of identity. Therefore, we will take top-level Moment Universals as ultimate and model them in Alloy as signatures (line 5 of Listing 15), as we did for Kinds, and the non-top-level ones will be modeled as subsignatures, just as SubKinds.

Moreover, from 13, p. 61 “A data type is a type whose instances are identified only by their value.”. Therefore, if a Datatype instance can be identified, then one of its types must provide a principle of identification for it. So, in the same way as for Moment Universals, we will take top-level Simple Datatypes as ultimate (being responsible in providing an unique identity principle for its instances) and model them in Alloy as signatures, and the non-top-level ones will be modeled as subsignatures.

Regarding Structured Datatypes, they are sets of n-tuples. Thus, it would be reasonable to model them as relations in Alloy. However, Alloy does not allows the creation of relations of relations. Therefore, if we model Structured Datatypes as relations, we would have to collapse high-order instances into flat ones. For example, an instance $((x,y),z)$ of an Structured Datatype SD_1, which is composed of another Structured Datatype SD_2 (which is responsible for the (x,y) instance) and a Simple Datatype SD_3 (which is responsible for the z instance), would be collapsed into a (x,y,z) instance, which corresponds to a different Structured Datatype SD_4, composed solely by Simple Datatypes.

Moreover, if we model Structured Datatypes as relations, in order to visualize an instance of a Structured Datatype as an attribute of a Classifier instance, we have to choose a privileged attribute of the Structured Datatype to be directly related to the Classifier instance, and the other attributes of the Structured Datatype will appear merely as labels in the relation.

We think that a better approach is to represent the instances of Structured Datatypes as atoms, in a way that the n-tuples are represented by atoms having relations with the constituents of the

n-tuples, which are their attributes. Therefore, we model Structured Datatypes just as Simple Datatypes, as sets of atoms (*i.e.*, Alloy signatures), but having relations with the Datatypes that are its attributes. So, if a Classifier has a Structured Datatype as an attribute, the atoms that are instances of the Classifier will be related to atoms that are instances of the Structured Datatype, in a way that the latter atoms will also have relations with another atoms representing its own attributes. This approach solves the visualization problem discussed above, as the Classifier instances will be related to the atoms that are instances of the Structured Datatypes, instead of being related to one of the constituents of the Structured Datatypes instances.

However, from 13, p. 61 we have that:

> "All copies of an instance of a data type and any instances of that data type with the same value are considered to be the same instance. Instances of a data type that have attributes (*i.e.*, is a structured data type) are considered to be the same if the structure is the same and the values of the corresponding attributes are the same. If a data type has attributes, then instances of that data type will contain attribute values matching the attributes." (13, p. 61)

As we model Simple Datatypes as atoms, those constraints automatically hold for them, because every atom is considered a distinct instance. Also, regarding Structured Datatypes, if we model them as relations, those constraints will automatically hold for them too, because, as n-tuples, their identity will be determined by its constituents. However, as we model Structured Datatypes as atoms, we have to guarantee that the following constraints will hold:

- The attributes of a StructuredDatatype instance cannot change from world to world. For example, consider the OntoUML model shown in Fig. 11, which models a «simpleDatatype» NaturalNumber; a «structuredDatatype» Birthday that has three NaturalNumber attributes (by means of «datatypeRelationship» relationships), namely day, month and year; and a «kind» Person that has an attribute age that is a NaturalNumber and an attribute birthday that is a Birthday. In an instance of this model, a Person John can change his age attribute from world to world (or time to time, in a temporal interpretation of worlds), but a (StructuredDatatype) Birthday instance "12/18/1982" (which is the birthday of John) cannot change its day attribute from 18 to 15, or, more generally, it cannot change any of its attributes without ceasing to be "12/18/1982".

- When two StructuredDatatype instances have the same values in its attributes, they have to be the same instance. For example, again referring to the OntoUML pictured in Fig. 11, when two (StructuredDatatype) Birthday instances α and β have the same attributes $day = 18$, $month = 12$ and $year = 1982$, they have to be the same "12/18/1982" instance, i.e., there can be at most one atom representing this instance. We name this property "canonicity".

In fact, the first constraint is guaranteed by the third additional invariant for the metaclass DatatypeRelationship, shown in page 70 and formalized in OCL in Listing 68, which constrains the Datatype Relationships that have StructuredDatatypes in its sources to have the value "true" in the meta-attribute isReadOnly of the Properties in its targets. Later in this section, we show how we model relationships that have read-only extremities in Alloy.

Now, we will better define the notion of canonicity in order to facilitate the understanding of the Alloy counterpart. First of all, SimpleDatatypes are canonical *a priori*. However, a StructuredDatatype D is canonical iff D is *locally canonical* and all its attributes are instances of Datatypes that are canonical. In other words, a StructuredDatatype D is canonical iff D is *locally canonical* and for all Datatype Relationships DR that have D in their source extremity, the Datatype connected in the target extremity of DR is canonical. Moreover, as all SimpleDatatypes are canonical, we can further refine this notion: A StructuredDatatype D is canonical iff D is *locally canonical* and for all Datatype Relationships DR that have D in their source extremity, if the Datatype connected in the target extremity of DR is a StructuredDatatype then it is canonical.

Definition 31 (Local Canonicity): By *local canonicity* we mean that a StructuredDatatype D is locally canonical iff for every x and y that are instances of D, if x has the same attributes than y, then x and y are the same instance. We can formalize this notion as $\mathit{Locally\_canonical}(\mathrm{D},\mathrm{S}_1,\ldots,\mathrm{S}_n,\mathrm{F})$, where D is the StructuredDatatype, $\mathrm{S}_1,\ldots,\mathrm{S}_n$ are the Datatype Relationships that have D in their source extremity, and F is the figure[7] function: $\mathit{Locally\_canonical}(\mathrm{D},\mathrm{S}_1,\ldots,\mathrm{S}_n,\mathrm{F}) \triangleq \forall x,y((\mathrm{D}(x) \land \mathrm{D}(y)) \rightarrow (\bigwedge_{i\leq n}(\forall z(((x,z) \in \mathrm{F}(\mathrm{S}_i)) \leftrightarrow ((y,z) \in \mathrm{F}(\mathrm{S}_i))))) \rightarrow (x = y)))$ ■

In the following, we reformulate our notion of canonicity in a non-recursive manner, in order to facilitate the formalization. One should notice that the recursive step in the previous definition of canonicity is used to guarantee that all StructuredDatatypes $\mathrm{D}^{\backprime}$ that are "reachable" from D, by means of Datatype Relationships, are locally canonical. Therefore, a StructuredDatatype D is canonical iff D is locally canonical and all StructuredDatatypes $\mathrm{D}^{\backprime}$ that are "reachable" from D, by means of Datatype Relationships, are locally canonical. This constraint is formalized in Definition 32.

Definition 32 (Canonicity): A StructuredDatatype D is canonical iff D is locally canonical

[7] We consider that a finitary relation L is defined by two mathematical objects: the ground of L, notated as G(L), and the figure of L, notated as F(L). G(L) is a sequence of k nonempty sets, $\mathrm{X}_1,\ldots,\mathrm{X}_k$, called the domains of the relation L. F(L) is a subset of the Cartesian product taken over the domains of L, that is, $\mathrm{F(L)} \subseteq \mathrm{X}_1 \times \ldots \times \mathrm{X}_k$. Therefore, L = (F(L),G(L)).

and for all Datatypes $\mathrm{D}^{\backprime}$ that are "reachable" from D by means of Datatype Relationships, if $\mathrm{D}^{\backprime}$ is a StructuredDatatype, then $\mathrm{D}^{\backprime}$ is locally canonical. More formally, a StructuredDatatype D is canonical iff D is locally canonical and for all the tuples ($\mathrm{D},\mathrm{D}^{\backprime}$) that are in the figure of the transitive closure of the meta-relation DatatypeRelationship, if $\mathrm{D}^{\backprime}$ is a StructuredDatatype, then $\mathrm{D}^{\backprime}$ is locally canonical. We can formalize this notion as $\mathit{Canonical}((\mathrm{D}_1,\mathrm{S}_{1_1},\ldots,\mathrm{S}_{1_{m_1}}),\ldots,(\mathrm{D}_i,\mathrm{S}_{i_1},\ldots,\mathrm{S}_{i_{m_i}}),\ldots,(\mathrm{D}_n,\mathrm{S}_{n_1},\ldots,\mathrm{S}_{n_{m_n}}),\mathrm{F})$, where D_1 is the target StructuredDatatype, D_i ($i > 1$) is a StructuredDatatype from a tuple (D_1,D_i) contained in the figure of the transitive closure of the meta-relation DatatypeRelationship, $\mathrm{S}_{i_1},\ldots,\mathrm{S}_{i_{m_i}}$ are the Datatype Relationships that have D_i in their source extremity, and F is the figure function: $\mathit{Canonical}((\mathrm{D}_1,\mathrm{S}_{1_1},\ldots,\mathrm{S}_{1_{m_1}}),\ldots,(\mathrm{D}_i,\mathrm{S}_{i_1},\ldots,\mathrm{S}_{i_{m_i}}),\ldots,(\mathrm{D}_n,\mathrm{S}_{n_1},\ldots,\mathrm{S}_{n_{m_n}}),\mathrm{F}) \triangleq \bigwedge_{i\leq n}\mathit{Locally\_canonical}(\mathrm{D}_i,\mathrm{S}_{i_1},\ldots,\mathrm{S}_{i_{m_i}},\mathrm{F})$ ■

One should notice that, from Definitions 31 and 32, if we guarantee that every StructuredDatatype D in an OntoUML model is locally canonical, then every D will also be canonical. In fact, this is a necessary and sufficient condition for all StructuredDatatypes to be concomitantly canonical. Therefore, is sufficient to impose only local canonicity to the StructuredDatatypes in the Alloy specification.

Let us exemplify the codification of Datatypes in Alloy by using the model shown in Fig. 11. The codification of this model in Alloy is shown in Listing 14. Lines 4 and 5 of Listing 14 shows the modeling of the SimpleDatatype NaturalNumber and the StructuredDatatype Birthday, respectively. Line 10 of Listing 14 shows the codification of the local canonicity constraint (Definition 31) in Alloy.

We model subtyping and GeneralizationSets (see Definitions 5, 6, 7) between Datatypes in the same way as for SubKinds.

As Categories are abstract, their instances are always instances of at least one of their non-abstract subtypes (otherwise, if they have no non-abstract subtypes, then they must have no instances at all), and as they are rigid, their instances never cease to be instances of them. Therefore, if a Category have subtypes (see Definition 4), then it can be modeled as a set that is the union of the instances of its subtypes. But, if we declare Categories as signatures, they will be automatically pairwise disjoint from all the other signatures. Therefore, we model a Category as a nullary function composed of a constant output that is the union of the signatures of all of its subtypes (see lines 10, 11 and 12 of Listing 15). We model subtyping for Categories by making the supertype as the union of the subtypes, for example if the Category C_1 is the supertype of the Categories $\mathrm{C}_2,\ldots,\mathrm{C}_n$, then the function "C1", representing C_1, will be the union of the functions "C2",...,"Cn", as shown in line 1 of Listing 11. If there is a GeneralizationSet (see Definitions

5, 6, 7) stating that $C_2, \ldots, C_n$ are disjoint, then we create a new fact stating that "C2",...,"Cn" are pairwise disjoint, as shown in line 2 of Listing 11.

Listing 11: Modeling Categories.

```
1 fun C1: univ {C2 + ... + Cn}
2 fact disjoint_categories {disj [C2,...,Cn]}
```

Also, as Phases, Roles, Mixins and RoleMixins are non-rigid (see Definition 21 for rigidity), the set of their instances may vary from world to world. Therefore, these classes are modeled within the "World" signature[8] as binary relations from worlds to sortals that are its supertypes (in the case of Phases and Roles) or its subtypes (in the case of Mixins and RoleMixins) and that are in the domain of quantification of that world (line 13 of Listing 15). The modeling of Phases and Roles (the anti-rigid sortals, see Definition 22 for anti-rigidity) is slightly different from the modeling of Mixins and RoleMixins (the non-sortals). The main difference is that, while Phases and Roles must be subtypes of sortals, Mixins and RoleMixins can only be supertypes of sortals.

For Phases and Roles, we model *subtyping* in two ways, regarding the nature of the *supertype* (*i.e.*, for a Phase or Role X and another class C, we model that X is a *subtype* of C in two different manners, regarding the nature of C). If the *superclass* is a rigid sortal RS, then we constrain the set of tuples of the relation representing the *subtype* (see Definition 4) to be a subset of the set of tuples of the relation representing the domain of quantification in which the second element is an instance of RS (see lines 15, 18, 19 and 20 of Listing 15). Otherwise, we declare the range of the Alloy relation as being the disjunction of the (non-rigid) *supertypes*, as shown in lines 16 and 21 of Listing 15.

As Mixins and RoleMixins are non-sortal abstract entities, we cannot model them in the same manner as we do for anti-rigid sortals. Therefore, we model *supertyping* for Mixins and RoleMixins in the following way: for a Mixin or RoleMixin X and another classes $\{C_1,\ldots,C_n\}$, we model that X is a *supertype* of $\{C_1,\ldots,C_n\}$ by constraining the range of the Alloy relation representing X to be the disjunction of the relations or signatures representing $\{C_1,\ldots,C_n\}$. When C_i is a rigid sortal, then it is represented as a signature Ci, so we will filter the Ci atoms that are in the domain of quantification of the worlds by using the following Alloy code: `Ci :>domain_of_quantification`. Otherwise, we use the name of the Ci relation alone. For example, consider the line 3 of Listing 12, which models that a Mixin M1 is the supertype of the Mixin M2 and of the Kind K. We also have to model that every instance of M2 or K is always an instance of M1. This constraint is shown in line 5 of Listing 12.

[8] The imported "world_structure" package specializes the World signature on order to build a temporal structure of worlds (line 1 of Listing 15).

Listing 12: Modeling Mixins and RoleMixins.

```
abstract sig World {
⋮
  M1: set (M2 + K:>domain_of_quantification),
}{
  all x: (M2 + K:>domain_of_quantification) | x in M1
⋮
}
```

Furthermore, in order to model GeneralizationSets of Phases and Roles, if the subclasses are disjoint (see Definition 5) and all of them have only one supertype, which is the common supertype, then we declare them together and use the keyword "disj" in the beginning of the line (see lines 15, 16, 18, 19 and 20 of Listing 15), otherwise, we declare the subclasses separately and for each disjoint GeneralizationSet we create disj[...] facts containing the disjoint subtypes. If the GeneralizationSets are complete (see Definition 6) and the superclass is not a rigid sortal (if it is declared as a relation), then we create a signature fact within the signature "World" stating that the set of tuples of the relation representing the superclass is equal to the union of the set of tuples of the relations representing the subclasses (see line 28 of Listing 15). If the subclasses are complete and the superclass is a rigid sortal, then we constrain the domain of quantification to only contain instances of the superclass that are also instances of at least one subclass (lines 26, 31, 33 and 35 of Listing 15).

Moreover, in order to model disjoint GeneralizationSets of Mixins or RoleMixins, if the subclasses are disjoint (see Definition 5) and all of them have the same set of subtypes, then we declare them together and use the keyword "disj" in the beginning of the line (see line 3 of Listing 13). For example, Listing 13 models that a Mixin M1 is the supertype of the Mixins M1 and M2, which are the supertypes of a Kind K. Otherwise, we declare the subclasses separately and for each disjoint GeneralizationSet we create disj[...] facts containing the disjoint subtypes. Finally, there is no need to model complete GeneralizationSets of Mixins and RoleMixins, because the modeling of their abstractness guarantees the completness.

Listing 13: Modeling disjoint GeneralizationSets of Mixins or RoleMixins.

```
1 abstract sig World {
2 ⋮
3   disj M2, M3: set K:>domain_of_quantification,
4   M1: set (M2 + M3),
}{
  all x: (M2 + M3) | x in M1
  all x: K:>domain_of_quantification | (x in M1) and (x in M2)
⋮
}
```

Furthermore, Phases are always defined in a partition set (see Definition 7) ‹$\mathrm{P}_1, \ldots, \mathrm{P}_\mathrm{n}$› constraining a sortal S (11, p. 103, *formulæ* 9, 10), and it is always possible (in the modal sense) for an instance x of S to become an instance of each P_i ($i \in 1, \ldots, n$) (11, p. 104) (Definition 9). Therefore, for any world w, if x is an instance of S in w, then x must be an instance of exactly one Phase of S in w and for each Phase P_i of S, there must exist a world in which x is an instance of P_i. In lines 27, 29, 32, 34 and 36 of Listing 15 we show how we model the last constraint.

Observe that these two last constraints (defined jointly in Definition 9) together imply anti-rigidity. Therefore, there is no need to model anti-rigidity constraints for Phases. However, we have to model anti-rigidity for Roles, but only for the ones that are not subtypes of another Roles or Phases. Because, from the Definition 22, when anti-rigidity is guaranteed for a class, then it is automatically guaranteed for all its subtypes. In other words, we only have to model anti-rigidity for the top level Roles. As in our running example all the Roles are subtypes of Phases, then there is no need to explicitly model anti-rigidity for them. However, for the sake of completeness, we show how we would model anti-rigidity in the commented lines 30 and 37 of Listing 15.

Regarding RoleMixins, their anti-rigidity is guaranteed by the anti-rigidity of their sortal subtypes. Because, (i) as RoleMixins are abstract, all of their instances are instances of at least one of their subtypes; and (ii) only anti-rigid classes can be subtypes of RoleMixins. Therefore, there is no need to pose a constraint to guarantee the anti-rigidity of the RoleMixins in the Alloy specifications.

We model OntoUML relationships in different forms regarding their modal implications. If an OntoUML binary relationship imply existential dependence (see Definition 22) from a rigid sortal RS to another Classifier, *i.e.*, if it is a Characterization, a Mediation, a Meronymic (*viz.* componentOf, memberOf, subCollectionOf or subQuantityOf) in which the values of the meta-attributes essential (see Definition 24) or inseparable (see Definition 28) are "true", or a DatatypeRelationship with a read-only target (an attribute of a StructuredDatatype, for example), then it is modeled as an Alloy relation within the "RS" signature (see lines 6, 7, and 8 of Listing 14; and lines 6 and 7 of Listing 15). If both extremities of a Meronymic relationship are existentially dependent on each other (*i.e.*, if the relationship is essential and inseparable), we model the relationship within the signature of the part add a signature fact in order to constrain the cardinality regarding the whole. Additionally, we have to create a signature fact within "World" to guarantee that if an instance a is existentially dependent on another instance b, then for each world w, if $a \in \mathcal{D}(w)$, then $b \in \mathcal{D}(w)$ (see line 38 of Listing 15).

Otherwise, the OntoUML relationships are modeled as ternary relations within the "World" signature. Thus, for each of these OntoUML relationships, different sets of tuples will exist in each world (see lines 14 and 15 of Listing 14). Additionally, if an OntoUML relationship R imply existential dependence from a non-rigid sortal NRS to another Classifier C, *i.e.*, if it is a Meronymic relationship R in which the values of the meta-attributes isImmutablePart (see Definition 25) or isImmutableWhole (see Definition 30) are "true", or a DatatypeRelationship with a read-only target, then we have to pose an additional constraint to guarantee that if an instance $rs1 \in \mathcal{D}(w)$ is such that *rs1*::NRS[9] in w and *rs1* is related to an instance $rs2 \in \mathcal{D}(w)$ (such that *rs2*::C in w) by an instance r of R in w, then in whatever world w' in which $rs1 \in \mathcal{D}(w')$ and *rs1*::NRS, then $rs2 \in \mathcal{D}(w')$ and *rs2*::C and *rs1* must be related to *rs2* by an instance r' of R in w' (see lines 39 and 40 of Listing 15), as stated in the Definitions 25 and 30.

Listing 14: The Alloy specification generated for the OntoUML model pictured in Fig. 11.

```
1 open world_structure[World]
2
3 sig Person {}
4 sig NaturalNumber {}
5 sig Birthday {
6   day: one NaturalNumber,
7   month: one NaturalNumber,
8   year: one NaturalNumber,
9 }{
10   all x,y: Birthday | ((x.@day = y.@day) and (x.@month =
       y.@month) and (x.@year = y.@year)) implies (x = y)
11 }
12 abstract sig World {
13   domain_of_quantification: some (Person),
14   age: set NaturalNumber one -> set Person:>
       domain_of_quantification,
15   birthday: set Birthday one -> set Person:>
```

[9] The symbol a::A means that a is an instance of A.

```
        domain_of_quantification,
16 }{
17   all x:Person:>domain_of_quantification | #age.x = 1
18   all x:Person:>domain_of_quantification | #birthday.x = 1
19 }
```

Subtyping, abstractness and GeneralizationSets between OntoUML relationships are modeled similarly as for non-rigid sortals, the only difference is that we will deal with ternary Alloy relations instead of binary ones. Regarding cardinalities, the Alloy language has some keywords for the most usual cardinalities, like "`set`", "`lone`", "`one`" and "`some`" meaning "0..*", "0..1", "1..1" and "1..*" respectively. In order to model single-tuple cardinalities of relationships, we can use these keywords on the declaration of a relation, as shown in lines 6, 7, 8, 14 and 15 of Listing 14, and also in the lines 22, 23 and 24 of Listing 15. In the case the relationship is declared within a signature that is its first domain (like the ones in lines 6 and 7 of Listing 15), then we may have to include a signature fact in order to constrain the cardinality of its first extremity, as show in lines 41 and 42 of Listing 15. In order to model general "m..n" cardinalities, we can use the dot operator "`.`" to navigate to a relation's extremity and the cardinality operator "#" to constrain the size of the set of elements in that extremity by writing inequalities. For example, if a relation R between A and B has an "m..n" cardinality constraint on the extremity B, we would write a fact like "`all a:A:>domain_of_quantification | (#(a.R)>= m)and (#( a.R)<= n)`" (see lines 17 and 18 of Listing 14). In order to model multiple-tuple cardinalities of a relationship R, one shall consider unfolding R in a Material Association, a Relator, some Mediation relationships and a Derivation relationship (11) (see Fig. 27), so every cardinality constraint will be treated as single-tuple.

Please note that we model no null minimum cardinality or "infinity" maximum cardinality (commonly represented as "*"). We decide to not model these cardinalities as they represent unconstrained minimum or maximum cardinalities. Also, although minimum cardinalities of "*" are theoretically possible, the Alloy Analyzer cannot handle them because it works with a limited number of atoms (see section 2.3 of chapter 2).

Regarding the shareability of Meronymic relationships, from the Definition 20, one can conclude that the only implication of setting the value of the meta-attribute isShareable to "false" in a relationship R is that the cardinality on the side of the whole must be exactly one. Therefore, non-shareability can be modeled solely by modeling cardinality constraints.

In order to model an n-ary Material Association M (whose tuples are in $C_1 \times \ldots \times C_n$), we model the Derivation and the Mediations within the signature of the respective Relator R (lines

6, 7 and 8 in Listing 15), and model M within the signature "World" (lines 22 in Listing 15). As Alloy is a first order language, then it does not allow the creation of tuples containing tuples. Therefore the Derivation relation will not be modeled as a relation between instances of R and M. Instead, we model it as an (n+1)-ary relation whose tuples are in $\mathrm{R}\times\mathrm{C}_1 \times \ldots \times\mathrm{C}_n$. We also constrain M (by means of a signature fact) in a way that for any world w, M is a subset of the set of Derivations of all the instances of R that are in $\mathcal{D}(w)$ (line 43 of Listing 15).

Since the most fundamental criteria for individuation are spatiotemporal and constrain instances of Sortal Universals to move on spatiotemporally continuous paths (68), we further constrain the generation of instances to produce only examples in which atoms have continuous existence, *i.e.*, from the point o view of a world w, if we have a world w' in w's past and a world w'' in w's future, it is not possible that an atom is in $\mathcal{D}(w')$ and in $\mathcal{D}(w'')$, but not in $\mathcal{D}(w)$. This constraint is posed in line 46 of Listing 15.

Moreover, as the accessibility relation is total in QS5, then the modal operators of possibility ($\Diamond$) and necessity ($\Box$) will take worlds in the set of all worlds ($\mathcal{W}$). Thus, in order to reduce the the computational complexity of analyzing the Alloy specification, we will modify the definitions of these modal operators to use $\mathcal{W}$ instead of the accessibility relation. For the same reasons, we will constrain every atom to be in the domain of quantification of some world, otherwise, Alloy Analyzer could generate atoms that would not be shown. This constraint is shown in line 47 of Listing 15.

The entire transformation presented in this section is implemented in ATL and is shown in appendix B.

5.6 A Case Study for the Transformation: Validating an OntoUML Model

In this section we will create a case study for the mappings from OntoUML to Alloy described in section 5.5 and their implementation in form of an ATL transformation, which is described in appendix B. Our case study will be based on the OntoUML model depicted in Fig. 27 (from section 4.6.4).

The Alloy specification automatically generated for the OntoUML model shown in Fig. 27 by the ATL transformation is shown in Listing 15.

Listing 15: The Alloy specification obtained automatically from the OntoUML model pictured in Fig. 27 by our ATL transformation (see appendix B).

```
open world_structure[World]
abstract sig Person {}
sig Man, Woman extends Person {}
sig Heart, Brain, Organization {}
sig Enrollment{
  student: one Person,
  school: one Organization,
  derived_study: student one -> one school
}
fun BiologicalOrgan: (Heart + Brain) {
  Heart + Brain
}
abstract sig World {
  domain_of_quantification: some (Person + Heart + Brain +
        Organization + Enrollment),
  disj LivingPerson, DeceasedPerson: set Person:>
      domain_of_quantification,
  disj Adult, Child, Teenager: set LivingPerson,
  Student: set LivingPerson,
  disj FunctionalHeart, NonfunctionalHeart: set Heart:>
      domain_of_quantification,
  disj FunctionalBrain, NonfunctionalBrain: set Brain:>
      domain_of_quantification,
  disj ActiveOrganization, ExtinctOrganization: set
      Organization:>domain_of_quantification,
  School: set ActiveOrganization,
  study: set Student some -> some School,
  c1: set FunctionalBrain one -> one LivingPerson,
  c2: set FunctionalHeart one -> lone LivingPerson
}{
  Person:>domain_of_quantification = LivingPerson +
      DeceasedPerson
  all x: Person | some w0,w1: World | (x in
      w0.@LivingPerson) and (x in w1.@DeceasedPerson)
  LivingPerson = Adult + Child + Teenager
  all x: LivingPerson | some w0,w1,w2: World | (x in
      w0.@Child) and (x in w1.@Teenager) and (x in
      w2.@Adult)
--all x: Student | some w: World | (x in
    w.@domain_of_quantification) and (x not in w.@Student)
  Heart:>domain_of_quantification = FunctionalHeart +
      NonfunctionalHeart
  all x: Heart | some w0,w1: World | (x in
      w0.@FunctionalHeart) and (x in w1.@NonfunctionalHeart
      )
  Brain:>domain_of_quantification = FunctionalBrain +
      NonfunctionalBrain
  all x: Brain | some w0,w1: World | (x in
      w0.@FunctionalBrain) and (x in w1.@NonfunctionalBrain
      )
  Organization:>domain_of_quantification =
      ActiveOrganization + ExtinctOrganization
  all x: Organization | some w0,w1: World | (x in
      w0.@ActiveOrganization) and (x in
      w1.@ExtinctOrganization)
--all x: School | some w: World | (x in
    w.@domain_of_quantification) and (x not in w.@School)
  all x: Enrollment:>domain_of_quantification | x.school
      in School and x.student in Student
  all x: Person, w0, w1: (@c1.x).Brain | (w0.@c1).x = (
      w1.@c1).x -- immutablePart.
  all x: Brain, w0, w1: (@c1.Person).x | x.(w0.@c1) = x.(
      w1.@c1) -- immutableWhole.
  all x: Student | some (student.x):>
      domain_of_quantification
  all x: School | some (school.x):>
      domain_of_quantification
  study = (Enrollment:>domain_of_quantification).
      derived_study
}
```

```
fact additional_facts {
  all w: World, x: (@next.w).domain_of_quantification | (x
     not in w.domain_of_quantification) => (x not in ((w.^
    next).domain_of_quantification))
  all x: (Person + Heart + Brain + Organization +
    Enrollment) | some w: World | x in
    w.domain_of_quantification
}
run{}
```

5.6.1 Generating Instances

In this subsection, we show the automatic generation of instances by the Alloy Analyzer tool. Fig. 35 depicts an instance of Listing 15 that is automatically generated by the Alloy Analyzer tool.

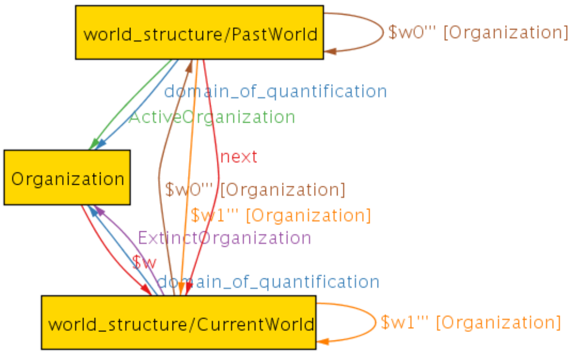
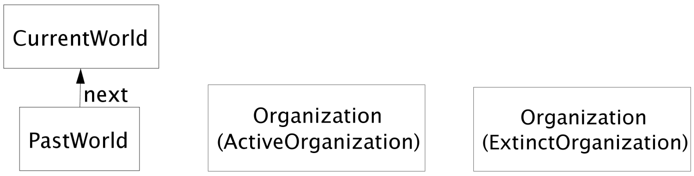


Figure 35: An instance for the Alloy specification depicted in Listing 15.

Although valid from a logic point of view, these automatically generated presentations are not suitable to be inspected and reasoned upon by the human modeler. Fortunately, the Alloy Analyzer allows the creation of visualization themes (66). Here, we take advantage of this feature by providing two visualization themes, one for visualizing the temporal ordering of worlds (shown in Listing 117 in appendix E) and the other to visualize the atoms by projecting them in each world (shown in Listing 118 in appendix E).

By applying these themes on the instance shown in Fig 35, we get Fig. 36. Fig. 36a shows the generated temporal ordering of worlds and Figs. 36b and 36c shows the atoms that are in the domain of quantification of each world.

As one can see, despite being a valid instance, the instance shown in Fig. 36 is of little


CurrentWorld
next
PastWorld
Organization (ActiveOrganization)
Organization (ExtinctOrganization)

(a) The temporal ordering of worlds.

(b) Instance at the past moment.

(c) Instance in the current moment.

Figure 36: Application of visualization themes on the instance shown in Fig. 35.

interest, as it only shows an active organization becoming an extinct organization in the current world.

The Alloy Analyzer creates first the simpler instances (the ones composed of fewer atoms) and then it reaches complex ones, by increasing the number of atoms per instances. Therefore, instead of visualizing every instance generated by the tool, in order to find representative ones for validation purposes, we can further qualify the type of instances we want the Alloy Analyzer to generate. For example, we will impose the generation of an instance having a person in at least one world, two disjoint hearts, at least one world in which there is a school, and one of each type of worlds (*viz.* past, counterfactual, current or future). This constraint is shown in Listing 16 and the generated instance is shown in Fig. 37.

Listing 16: Constraining the generation of instances.

```
run {
  (#Person = 1) and (#Heart = 2) and (#School >= 1) and (
      #CounterfactualWorld = 1) and (#PastWorld = 1) and (
      #FutureWorld = 1)
} for 4
```

After applying the themes in the instance shown in Fig. 37, we get the Figs. 38, which shows the temporal ordering of worlds, and 39, which shows the atoms that are in the domain of quantification of each world. In Fig. 39, the application of the theme shown in Listing 118 in appendix E makes the instances of the Kinds Brain or Heart to be represented as trapeziums, the instances of the Kind Person to be represented as ellipses, the instances of the Kind Organization to be represented as rectangles and the instances of the Relator Enrollment to be represented as hexagons. Also, instantiation of Categories (*e.g.*, BiologicalOrgan), Phases (*e.g.*, LivingPerson, DeceasedPerson, Child, Teenager, Adult, ActiveOrganization, ExtinctOrganization, FunctionalBrain, NonfunctionalBrain, FunctionalHeart and NonfunctionalHeart) and Roles (*e.g.*, Student and School) are represented as labels.

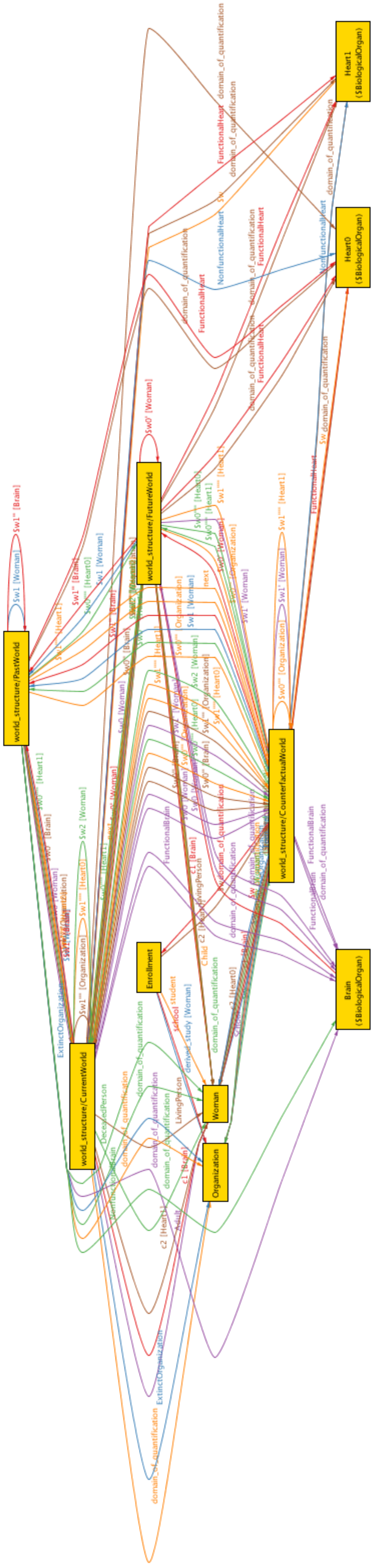

world_structure/PastWorld
world_structure/FutureWorld
world_structure/CurrentWorld
world_structure/CounterfactualWorld
Enrollment
Woman
Organization
Brain ($BiologicalOrgan)
Heart0 ($BiologicalOrgan)
Heart1 ($BiologicalOrgan)

Figure 37: A second attempt to generate a feasible instance.

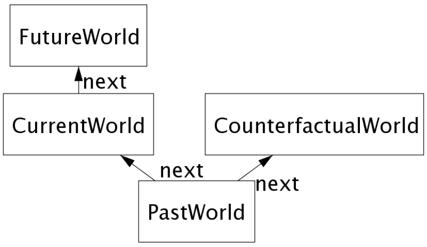
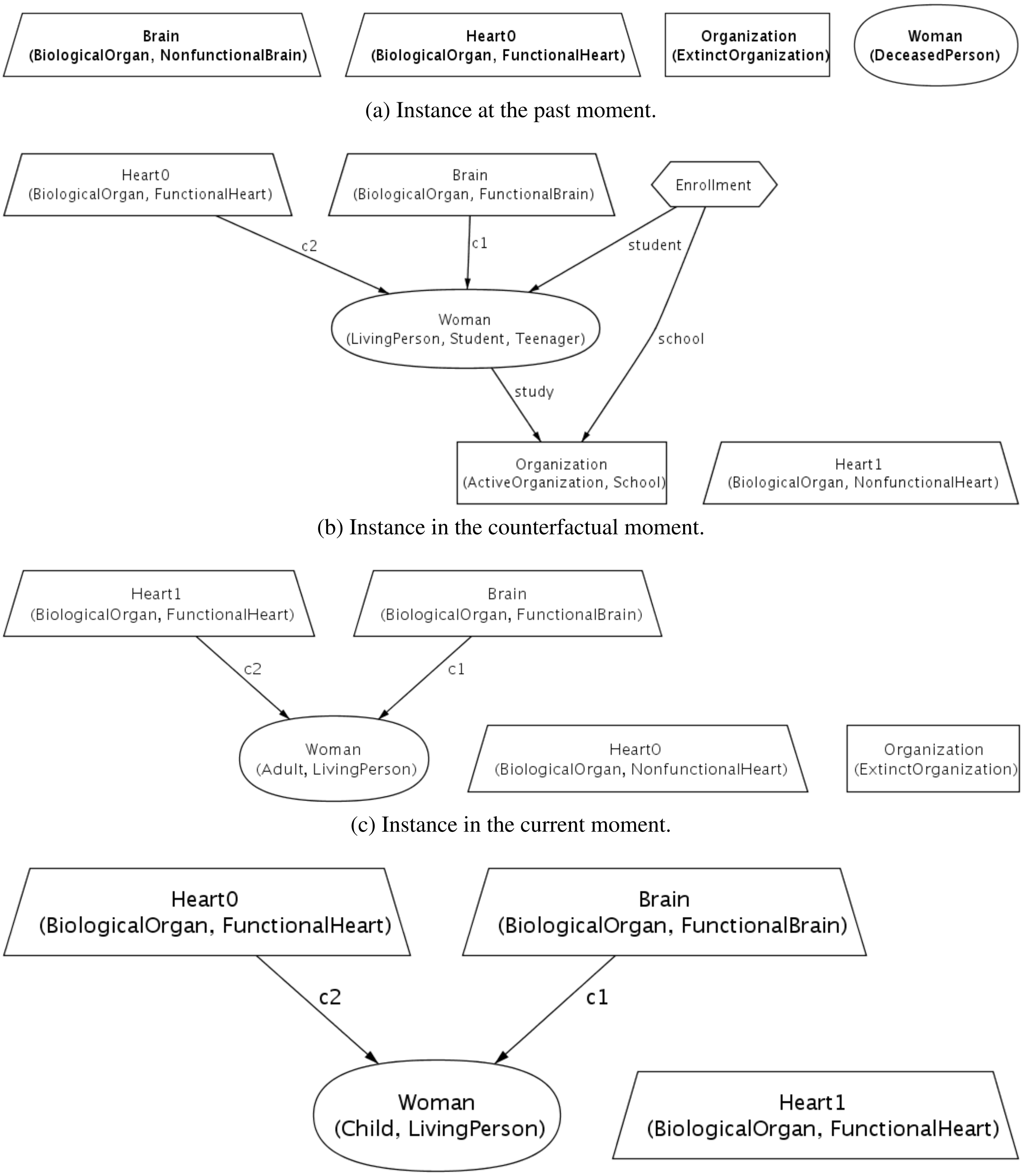


Figure 38: The temporal ordering of worlds.


Brain
(BiologicalOrgan, NonfunctionalBrain)
Heart0
(BiologicalOrgan, FunctionalHeart)
Organization
(ExtinctOrganization)
Woman
(DeceasedPerson)

(a) Instance at the past moment.

Heart0
(BiologicalOrgan, FunctionalHeart)
Brain
(BiologicalOrgan, FunctionalBrain)
Enrollment
c2
c1
student
Woman
(LivingPerson, Student, Teenager)
school
study
Organization
(ActiveOrganization, School)
Heart1
(BiologicalOrgan, NonfunctionalHeart)

(b) Instance in the counterfactual moment.

Heart1
(BiologicalOrgan, FunctionalHeart)
Brain
(BiologicalOrgan, FunctionalBrain)
c2
c1
Woman
(Adult, LivingPerson)
Heart0
(BiologicalOrgan, NonfunctionalHeart)
Organization
(ExtinctOrganization)

(c) Instance in the current moment.

Heart0
(BiologicalOrgan, FunctionalHeart)
Brain
(BiologicalOrgan, FunctionalBrain)
c2
c1
Woman
(Child, LivingPerson)
Heart1
(BiologicalOrgan, FunctionalHeart)

(d) Instance in a future moment.

Figure 39: Atoms projected by worlds.

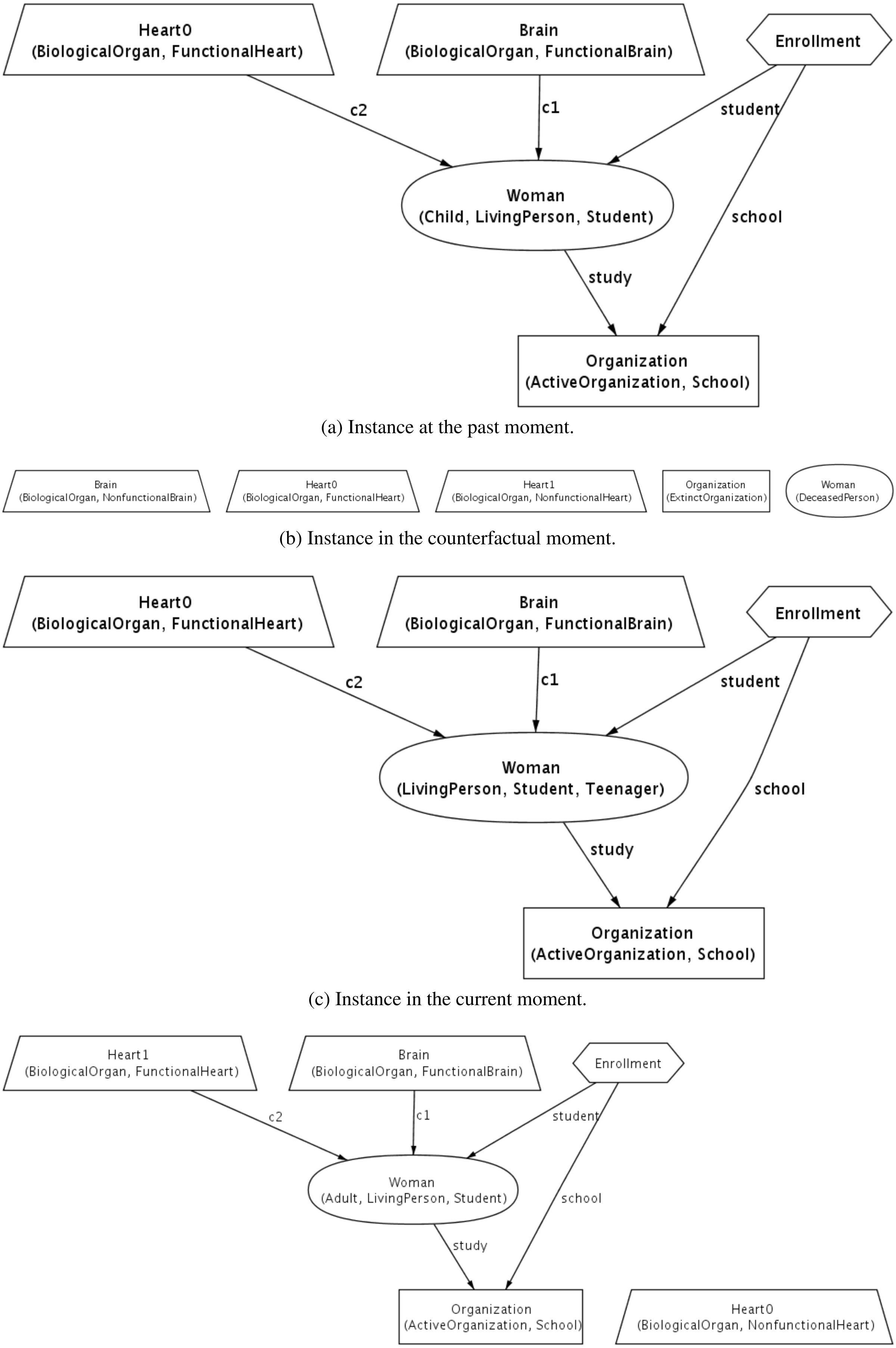
As one can see, in the past world, the woman is deceased, but in the current world she is an adult, in a counterfactual one she could be a teenager and in a future one she may even become a child! As OntoUML does not contemplate the explicitly modeling of the temporal ordering of Phases, we will introduce this ordering directly in the Alloy counterpart specification as a new fact, as show in the Listing 17.

Listing 17: Modeling the ordering of the Phases.

```
fact an_ordering_for_the_phases{
  all x:Person, w0: World, w1:w0.next | (x in
      w0.DeceasedPerson) => (x not in w1.LivingPerson)
  all x:Person, w0: World, w1:w0.next | ((x in w0.Child)
      => (x not in w1.Adult)) and ((x in w0.Teenager) => (x
       not in w1.Child)) and ((x in w0.Adult) => ((x not in
       w1.Child) and (x not in w1.Teenager)))
  all x:Heart, w0: World, w1:w0.next | (x in
      w0.NonfunctionalHeart) => (x not in
      w1.FunctionalHeart)
  all x:Brain, w0: World, w1:w0.next | (x in
      w0.NonfunctionalBrain) => (x not in
      w1.FunctionalBrain)
  all x:Organization, w0: World, w1:w0.next | (x in
      w0.ExtinctOrganization) => (x not in
      w1.ActiveOrganization)
}
```

Now, the generated instances will be well ordered regarding the Phases LivingPerson-/DeceasedPerson, Child/Teenager/Adult, FunctionalBrain/NonfunctionalBrain, Functional-Heart/NonfunctionalHeart and ActiveOrganization/ExtinctOrganization. An example is pictured in Fig. 40, which has the same temporal ordering of worlds that is shown in Fig. 38.

The instance depicted by Figs. 40 and 38 exemplifies some important constraints like the rigidity (Definition 21) of the Kinds and Categories, *e.g.*, the woman never ceases to be an instance of the Kind Person while she is in a domain of quantification of a world); the anti-rigidity (Definition 22) of the Phases, *e.g.*, regarding the Phases Child, Teenager and Adult, for every world w in which the woman is in one of these Phases, there is a world w' in which she is not in that Phase; the anti-rigidity of the Roles, *e.g.*, the Role Student, in which for every world w in which the woman plays this Role, there is a world w' in which she does not play


Heart0
(BiologicalOrgan, FunctionalHeart)
Brain
(BiologicalOrgan, FunctionalBrain)
Enrollment
c2
c1
student
Woman
(Child, LivingPerson, Student)
school
study
Organization
(ActiveOrganization, School)
Brain
(BiologicalOrgan, NonfunctionalBrain)
Heart0
(BiologicalOrgan, FunctionalHeart)
Heart1
(BiologicalOrgan, NonfunctionalHeart)
Organization
(ExtinctOrganization)
Woman
(DeceasedPerson)
Heart0
(BiologicalOrgan, FunctionalHeart)
Brain
(BiologicalOrgan, FunctionalBrain)
Enrollment
c2
c1
student
Woman
(LivingPerson, Student, Teenager)
school
study
Organization
(ActiveOrganization, School)
Heart1
(BiologicalOrgan, FunctionalHeart)
Brain
(BiologicalOrgan, FunctionalBrain)
Enrollment
c2
c1
student
Woman
(Adult, LivingPerson, Student)
school
study
Organization
(ActiveOrganization, School)
Heart0
(BiologicalOrgan, NonfunctionalHeart)

(a) Instance at the past moment.

(b) Instance in the counterfactual moment.

(c) Instance in the current moment.

(d) Instance in a future moment.

Figure 40: Instance with ordered Phases.

this Role; the relational dependence (Definition 22) of the Roles are validated, *e.g.*, for the Role Student, in which the woman can only play this Role while related to an instance of School by an instance of Enrollment. Some well known conceptual modeling primitives are also validated, such as the abstractness for Person (its instances have to be instances of Man or Woman), and the disjointness and completeness for all Phase partitions.

Also, this instance illustrates the immutability of both part and whole (Definition 25 and 30) regarding the woman and her brain (she never changes her brain while alive and the brain never changes its whole when functional), depicted by the "{immutable part, immutable whole}" tag in the relationship between Person and Brain (Fig. 27). One can notice that in a future world, the woman will change her heart (maybe she will undergo a heart transplant), while her old heart will became nonfunctional. This behaviour is totally acceptable, as she is generically dependent on the Kind Heart.

Finally, this instance also satisfies the constraint that imposes that if an atom instantiates a Phase in some world, then it must possibly instantiate every Phase in that Phase partition.

5.7 Conclusions

In this chapter, in section 5.1 we exemplify the importance of a tool for generating instances in the validation phase, when the modeler can detect unintended behaviours and then correct the model.

Our approach is based on the transformation of OntoUML models into Alloy specifications. As shown is section 5.2, the product of this transformation is an Alloy specification that can be fed into the Alloy Analyzer to generate instances that respect UFO's (modal) axioms. Furthermore, we believe that the analysis of a well-chosen set of these instances can improve the modeler's confidence in the validity of the model.

In section 5.3, in order to facilitate the understanding of the dynamics of creation and destruction of OntoUML instances within worlds, we revisit the possibilist formalization of some OntoUML and UFO concepts (11, 55) in an actualist quantified modal logic QS5 with a varying domain of quantification.

In section 5.4, we define our Kripke structure and give it a superficial temporal meaning by classifying the worlds into four disjoint categories, *viz.* CurrentWorld, PastWorld, FutureWorld and CounterfactualWorld, and creating a partial order relation "next" between them, as well as some constraints to guarantee that the ordering between worlds will be a temporal one. In

Listing 10, we show how this structure was modeled in Alloy.

In section 5.5 we explain how we map OntoUML Classes, Relations, Datatypes, Generalizations and GeneralizationSets to Alloy by mapping the definitions elaborated in section 5.3 to Alloy modeling patterns.

Finally, in section 5.6 we create a case study for the mappings from OntoUML to Alloy described in section 5.5. By transforming the model depicted in Fig. 27 to an Alloy specification, we show our ATL transformation in action, using many of the mappings described in section 5.5.

Also, we show an instance that exemplifies some very important concepts of OntoUML, like rigidity (Definition 21), anti-rigidity (Definition 22), *etc.*, as well as some well known conceptual modeling primitives, such as disjointness and completeness (Definitions 5, 6 and 7).

6 Discussion & Final Considerations

In this chapter, we pose our original contributions (section 6.1), a list of publications that resulted from the present work (section 6.2), some difficulties we had along our research (section 6.3), some related work (section 6.4), final considerations (section 6.5), and the future work (section 6.6).

6.1 Original Contributions

Throughout this thesis, we have defined the abstract syntax of OntoUML in Ecore (section 4.2), formalized all the OntoUML syntactical constraints in OCL (appendix A), defined the OntoUML concrete syntax by using GMF technologies (section 4.4) and built a graphical editor that is capable of automatic checking of the OntoUML syntax constraints and automatic derivation of information from the models (section 4.5).

We also modeled a Kripke structure in Alloy (section 5.4), built a mapping from OntoUML models to Alloy specifications (section 5.5), and implemented this mapping as an automatic ATL transformation (appendix B), so one can automatically generate model instances in order to validate OntoUML models.

Therefore, we believe that the primary contribution of the present thesis is to build a graphical editor for OntoUML that:

- Allows the creation of conceptual models and ontologies graphically, in a simple way;
- Automatically verifies models (*i.e.*, check the OntoUML syntax constraints in models), when suitable;
- Allows the modeler to start syntactic checks manually, when he/she deems suitable;
- Informs the reason why a model is syntactically invalid in a way the modeler understands what is wrong, so he/she can figure out how to fix it;

- Automatically derives information from the models in specific contexts, saving the user from modeling information that could be automatically inferred;
- Allows the generation of model instances with the purpose of improving the modeler's confidence in the validity of the model.

As secondary contributions, we have produced the following MDE artifacts:

- An OntoUML metamodel in Ecore, with derived meta-relations implemented as OCL expressions;
- A set of OCL expressions formalizing the automatic initialization or modification of some meta-attributes' values, *e.g.*, OCL expressions for the calculation of the values of the upper cardinality constraint for both association ends of Material Associations and the lower and upper cardinality constraints of the source association end of Derivation relationships (11, p. 331, Figs. 8-10);
- A set of OCL expressions formalizing the OntoUML syntactical constraints;
- A GMF definition of the OntoUML concrete syntax;
- An actualist logical formalization of some UFO concepts;
- A categorization of worlds in a common sense temporal structure;
- An Alloy specification of the QS5 world structure with our defined temporal types and ordering;
- A transformation specification from OntoUML models to Alloy specifications.

6.2 Publications

The research documented in this thesis is also reported in a series of articles. The model building and verification capabilities of the OntoUML Editor, as well as its architecture, are reported in 69.

Moreover, the tool for assisting model validation, its architecture and transformation patterns are reported in 70, 71, 72, 73.

6.3 Difficulties

The main difficulties during the development of this thesis were due to the scarcity and low quality of the EMF and GMF documentation, and the instability of those Eclipse plug-ins. The GMF tutorials are extremely simplistic, which makes arid their use in robust projects.

The transformations realized in the Ecore metamodel for the generation of Java code were not capable of generating a functional graphical editor for OntoUML. Therefore, a lot of essential functionalities had to be implemented directly in Java, which was an arduous task, mainly due to the almost inexistence of documentation regarding the Java code generated by the EMF and GMF frameworks.

Another difficulty occurred in the OCL specification of the EOperations and derived EAttributes and EReferences in the Ecore OntoUML metamodel, because the EMF framework does not have a native support for the OCL language, being necessary the use of JET templates to enable the parsing of OCL expressions. Therefore, no syntactical verification is done during the specification of OCL expressions, no debug message is shown and faults in these expressions can only be detected after the realization of all the EMF and GMF transformations, by running the OntoUML Editor.

Also, files that should be always synchronized are constantly not synchronized, which leads to errors that are not alerted by the Eclipse plug-ins. An example is the synchronization of Ecore and Genmodel files.

As stated in section 4.5.1, the Bug 138179 (60) frustrated a lot the development of OntoUML Editor. Due to this bug, in order to be able to automatically create the visualizations of the Properties connected to a relationship (for example, cardinalities and role names), we had to include additional meta-attributes and meta-relations in the OntoUML metamodel as well as to customize the Java code generated by the EMF and GMF plug-ins.

Finally, we had no difficulties using the ATL Eclipse plug-in, as the ATL documentation is very rich and its Eclipse plug-in is stable. The same is valid for the Alloy documentation and the Alloy Analyzer tool.

6.4 Related Work

As far as we know, there is no other tool that supports the construction of syntactically correct OntoUML models. However, there are other editors that support philosophically well-founded languages and methodologies such as OntoClean (74, 75), as well as tools based on upper-level ontologies as Suggested Upper Merged Ontology[1] (SUMO), Standard Upper Ontology[2] (SUO) and the Differential Semantics theory. For instance, Protégé (76) is a free open-source tool that supports OntoClean and SUMO. Automatic Evaluation of ONtologies[3] (AEON) is an open-source tool, which allows applying OntoClean to evaluate ontologies. Visual Ontology Modeler (77) is an editor that includes a library of ontologies that represent SUO. Differential Ontology Editor[4] (DOE) is a freeware ontology editor that allows the user to build ontologies according to the Differential Semantics theory. Sigma (78) is a free open-source knowledge engineering environment for theories in FOL, which is optimized for SUMO.

Regarding model validation, several approaches in literature aim at assessing whether conceptual models comply with their intended conceptualizations. Although many approaches (*e.g.*, (79) and (80)) focus on analysis of behavioural UML models, we are primarily concerned with structural models and thus refrain further analysis of behavioural-focused work.

A prominent example is the UML Specification Environment[5] (14) (USE) tool. The tool is able to indicate whether instances of a UML class diagram respect constraints specified in the model through OCL. Differently from our approach, which is based on the automatic creation of example world structures, in USE the modeler must specify sequences of snapshots in order to gain confidence on the quality of the model (either through the user interface or by specifying sequences of snapshots in a tool-specific language called A Snapshot Sequence Language (ASSL)). Since no modal meta-property of classifiers is present in UML, this tool does not address modal aspects and validates constraints considering only a sole snapshot.

Finally, the approaches of 81 and 82[6] are similar to ours in that they translate UML class diagrams to Alloy. However, both of them translate all classes into Alloy signatures, which suggests that no dynamic classification is possible in these approaches. Similarly to our approach, Anastasakis *et al.* in 82 implements a model transformation using model-driven techniques to automatically generate Alloy specifications, while 81 relies on manual translation to Alloy. Similar to USE, 81 focuses on analysis and constraint validation on single snapshots. 82 introduces a notion of state transition but still does not address the modal aspects of classes since those are not part of UML.

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

[2] http://suo.ieee.org.

[3] http://ontoware.org/projects/aeon.

[4] http://homepages.cwi.nl/~troncy/DOE.

[5] http://www.db.informatik.uni-bremen.de/projects/USE.

[6] The UML2Alloy tool is available at http://www.cs.bham.ac.uk/~bxb/UML2Alloy.

6.5 Final Considerations

The need for using ontologically well-founded languages for conceptual modeling, in general, and domain ontologies, in particular, has increasingly been recognized in the literature. This is often a result of interoperability concerns and the unsuitability of lightweight representation languages in addressing these issues. Despite that, those languages are still not broadly adopted in practice. One of the main reasons is the need for high-level expertise in handling the philosophical concepts underlying them. Indeed, the dissemination of formal method techniques requires convincing industries and standardization bodies that such techniques in fact can improve development. In this way, design support tools are one of the key resources to foster their adoption in practice (83).

In this thesis, we present an Eclipse-based graphical editor that aims at fulfilling the gap of tool support for one particular theoretically well-founded representation language, namely, OntoUML. Underlying this editor there is an implementation of the OntoUML metamodel proposed in 11 by using MDA technologies, in particular, the OMG MOF and OCL. Moreover, by representing UFO's categories and axiomatization in the language metamodel, the complexity of these foundational issues is hidden from the user while still constraining him to produce ontologically sound models.

Furthermore, regardless of the (lack of) ontological commitment of the modeling language, a mature approach to conceptual modeling requires modelers to gain confidence on the quality of the models they produce, assessing whether those models express as accurately as possible an intended conceptualization. This thesis contributes to that goal, by providing tools to support the validation of the modal properties of a conceptual model in OntoUML.

Following a model-driven approach, we have defined and automated a transformation of OntoUML models into Alloy specifications. The generated Alloy specifications are fed into the Alloy Analyzer to create temporal world structures that show the possible dynamics of object creation, classification, association and destruction as defined in the model. The snapshots in this world structure confront a modeler with states-of-affairs that are deemed admissible by the model. This enables modelers to detect unintended states-of-affairs and take the proper measures to rectify the model. We believe that the example world structures support a modeler in the validation process, especially since it reveals how state-of-affairs evolve in time and how they may eventually evolve (revealing alternative scenarios implied by the model).

As a final remark, the promotion of a language such as OntoUML for domain engineering does not eliminate the need for codification languages such as Web Ontology Language (10)

(OWL), DLR$_{us}$ (84), Alloy or Frame Logic (5) (F-Logic), to cite just a few examples. In pace with the meaning independence principle defended by Guizzardi and Halpin (85), we adopt the view that those classes of languages are (and should be) meant to be used for different purposes and in different phases on an ontology engineering process.

6.6 Future Work

As future work, we aim at performing empirical studies about (i) the tool usability, and (ii) the effectiveness of this approach of validation based on simulations, taking into account scalability issues.

Moreover, we intend to incorporate support for domain constraints in our approach, *e.g.*, including OCL constraints in an OntoUML model. This will require (i) an extension on the OntoUML syntax verification method implemented in the editor in order to include the verification of the OCL syntactical constraints, and (ii) transforming those OCL constraints into Alloy specifications, in order to guarantee that the constraints are satisfied in all instances generated by the Alloy Analyzer tool.

Further, we intend to work on methodological support for the validation process, proposing guidelines for modelers to select relevant world structures. We will aim for an interactive approach in which a modeler can select which of the alternative scenarios to consider. We believe that this may help pruning the branches in the world structure keeping the size of this structure manageable.

Ideally, by exploring visualization techniques, we could use the instances generated by the Alloy Analyzer as example scenarios to be exposed to the stakeholders of the conceptual model (such as domain experts) in order to validate whether their conceptualization has been captured accurately by the modeler.

Bibliography

1 BÉZIVIN, J.; JOUAULT, F.; ROSENTHAL, P.; VALDURIEZ, P. Modeling in the large and modeling in the small. In ASSMANN, U.; AKSIT, M.; RENSINK, A. (Ed.). *MDAFA*. Springer, 2004. (Lecture Notes in Computer Science, vol. 3599), p. 33–46. ISBN 3-540-28240-8. Available from Internet: <http://dx.doi.org/10.1007/11538097_3>. Cited 01/13/2010.

2 JOUAULT, F.; ALLILAIRE, F.; BÉZIVIN, J.; KURTEV, I.; VALDURIEZ, P. ATL: a QVT-like transformation language. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM, 2006. p. 719–720. ISBN 1-59593-491-X. Available from Internet: <http://doi.acm.org/10.1145/1176617.1176691>. Cited 10/04/2009.

3 DEAN, D.; GERBER, A.; MOORE, B.; VANDERHEYDEN, P.; WAGENKNECHT, G. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks, 2004. 250 p. ISBN 0738453161. Available from Internet: <http://www.redbooks.ibm.com/abstracts/sg246302.html>. Cited 08/17/2009.

4 Meta-Object Facility (MOF) Core Specification, Version 2.0. [S.l.], January 2006. 76 p. Available from Internet: <http://www.omg.org/spec/MOF/2.0>. Cited 08/17/2009.

5 KIFER, M.; LAUSEN, G.; WU, J. Logical foundations of object-oriented and frame-based languages. *Journal of the Association for Computing Machinery (ACM)*, vol. 42, p. 741–843, 1990.

6 Institut national de recherche en informatique et en automatique (INRIA). *KM3: Kernel MetaMetaModel Manual version 0.3*. Nantes, France, August 2005.

7 MILLER, J.; MUKERJI, J. (Ed.). *MDA Guide Version 1.0.1*. [S.l.], June 2003. 62 p. Available from Internet: <http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf>. Cited 08/17/2009.

8 Object Constraint Language, Version 2.0. [S.l.], May 2006. 232 p. Available from Internet: <http://www.omg.org/spec/OCL/2.0>. Cited 08/17/2009.

9 FIRESMITH, D.; HENDERSON-SELLERS, B.; GRAHAM, I. *OPEN modeling language (OML) reference manual*. New York, NY, USA: Cambridge University Press, 1998. ISBN 0-521-64823-8.

10 World Wide Web Consortium (W3C). *OWL Web Ontology Language Guide*. Available from Internet: <http://www.w3.org/TR/owl-guide>. Cited 08/20/2009.

11 GUIZZARDI, G. *Ontological foundations for structural conceptual models*. Thesis (Ph.D.) — University of Twente, Enschede, The Netherlands, Enschede, October 2005. Available from Internet: <http://doc.utwente.nl/50826/>. Cited 01/13/2009.

12 OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.2. [S.l.], February 2009. 226 p. Available from Internet: <http://www.omg.org/spec/UML/2.2/Infrastructure>. Cited 08/17/2009.

13 OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.2. [S.l.], February 2009. 740 p. Available from Internet: <http://www.omg.org/spec/UML/2.2-/Superstructure>. Cited 08/17/2009.

14 GOGOLLA, M.; BÜTTNER, F.; RICHTERS, M. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming Special Issue on Experimental Software and Toolkits*, vol. 69, no. 1-3, p. 27–34, 2007.

15 MOF 2.0/XMI Mapping, Version 2.1.1. [S.l.], December 2007. 120 p. Available from Internet: <http://www.omg.org/spec/XMI/2.1.1>. Cited 10/04/2009.

16 MYLOPOULOS, J. Conceptual modeling, databases, and case: An integrated view of information systems development. In ____. Chichester: John Wiley & Sons, 1992. cap. Conceptual Modeling and Telos, p. 49–68.

17 GUIZZARDI, G. On ontology, ontologies, conceptualizations, modeling languages, and (meta)models. In VASILECAS, O.; EDER, J.; CAPLINSKAS, A. (Ed.). *DB&IS*. Amsterdam: IOS Press, 2006. (Frontiers in Artificial Intelligence and Applications, vol. 155), p. 18–39. ISBN 978-1-58603-715-4.

18 DEGEN, W.; HELLER, B.; HERRE, H.; SMITH, B. GOL: Toward an axiomatized upper-level ontology. In *Proceedings of the International Conference on Formal Ontology in Information Systems*. Ogunquit, Maine, USA: ACM New York, NY, USA, 2001. (Formal Ontology in Information Systems, 2), p. 34–46. Available from Internet: <http://dx.doi.org/10.1145/505168.505173>. Cited 08/17/2009.

19 GUARINO, N.; GUIZZARDI, G. In the defense of ontological foundations for conceptual modeling. *Scandinavian Journal of Information Systems*, vol. 18, no. 1, p. 115–126, 2006. ISSN 0905-0167. Invited Paper.

20 GUIZZARDI, G.; WAGNER, G.; GUARINO, N.; SINDEREN, M. van. An ontologically well-founded profile for uml conceptual models. In *Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE)*. Latvia: Springer-Verlag Berlin / Heidelberg, 2004. (Lecture Notes in Computer Science (LNCS), Volume 3084/2004), p. 112–126. ISBN 3-540-22151-4. Available from Internet: <http://dx.doi.org/10.1007/b98058>. Cited 08/17/2009.

21 GONÇALVES, B.; GUIZZARDI, G.; PEREIRA.FILHO, J. G. An electrocardiogram (ECG) domain ontology. In GUIZZARDI, G.; FARIAS, C. (Ed.). *Proceedings of the 2nd Workshop on Ontologies and Metamodels for Software and Data Engineering (WOMSDE), 22nd Brazilian Symposium on Databases (SBBD)/21st Brazilian Symposium on Software Engineering (SBES)*. João Pessoa, Brazil: [s.n.], 2007. p. 68–81. Available from Internet: <http://www.lbd.dcc.ufmg.br:8080/colecoes/womsde/2007/006.pdf>. Cited 09/21/2010.

22 OLIVEIRA, F.; ANTUNES, J.; GUIZZARDI, R. S. S. Towards a collaboration ontology. In GUIZZARDI, G.; FARIAS, C. (Ed.). *Proceedings of the 2nd Workshop on Ontologies and Metamodels for Software and Data Engineering (WOMSDE), 22nd*

Brazilian Symposium on Databases (SBBD)/21st Brazilian Symposium on Software Engineering (SBES). João Pessoa, Brazil: [s.n.], 2007. p. 97–108. Available from Internet: <http://www.lbd.dcc.ufmg.br:8080/colecoes/womsde/2007/008.pdf>. Cited 09/21/2010.

23 The Eclipse Foundation. *Ecore*. Available from Internet: <http://download.eclipse.org-/modeling/emf/emf/javadoc/2.5.0/org/eclipse/emf/ecore/package-summary.html>. Cited 08/17/2009.

24 The Eclipse Foundation. *Eclipse*. Available from Internet: <http://www.eclipse.org>. Cited 08/17/2009.

25 JACKSON, D. *Software abstractions : logic, language, and analysis*. [S.l.]: MIT Press, 2006. ISBN 978-0-262-10114-1.

26 Alloy Community. 2009. Available from Internet: <http://alloy.mit.edu/community>. Cited 08/17/2009.

27 JACKSON, D. Alloy: a lightweight object modelling notation. *Transactions on Software Engineering and Methodology (TOSEM)*, ACM, New York, NY, USA, vol. 11, no. 2, p. 256–290, 2002. ISSN 1049-331X.

28 PRIEST, G. *An introduction to non-classical logic*. Cambridge: Cambridge University Press, 2001. 00023609 GBA1-19206 Graham Priest. Includes bibliographical references and index.

29 HUGHES, G. E.; CRESSWELL, M. J. *A Companion to Modal Logic*. London: Routledge and Kegan Paul, 1985.

30 Stanford Encyclopedia of Philosophy. Available from Internet: <http://plato.stanford.edu-/entries/actualism/possibilism>. Cited 12/23/2009.

31 Stanford Encyclopedia of Philosophy. Available from Internet: <http://plato.stanford.edu-/entries/actualism>. Cited 12/23/2009.

32 FITTING, M.; MENDELSOHN, R. L. *First-order modal logic*. Norwell, MA, USA: Kluwer Academic Publishers, 1999. ISBN 0-7923-5334-X.

33 SCHMIDT, D. C. Model-driven engineering. *IEEE Computer*, vol. 39, no. 2, February 2006. Available from Internet: <http://www.truststc.org/pubs/30.html>. Cited 01/13/2010.

34 DALGARNO, M.; FOWLER, M. UML vs. Domain-Specific Languages. Martinig & Associates, vol. 16, no. 2, 2006. ISSN 1661-402X. Available from Internet: <http://www.methodsandtools.com/mt/download.php?summer08>. Cited 12/23/2009.

35 REIS, G. D.; JÄRVI, J. What is generic programming? In *Proceedings of the First International Workshop of Library-Centric Software Design (LCSD '05). An OOPSLA '05 workshop*. [S.l.: s.n.], 2005.

36 MAINTAINERS. Available from Internet: <http://www.generic-programming.org>. Cited 12/23/2009.

37 STEINBERG, D.; BUDINSKY, F.; PATERNOSTRO, M.; MERKS, E. *EMF: Eclipse Modeling Framework 2.0*. [S.l.]: Addison-Wesley Professional, 2009. ISBN 0321331885.

38 BÉZIVIN, J.; GÉRARD, S.; MULLER, P. A.; RIOUX, L. MDA components: Challenges and Opportunities. In . [S.l.: s.n.].

39 Syntropy Limited. Available from Internet: <http://www.syntropy.co.uk/syntropy>. Cited 10/04/2009.

40 WARMER, J. B.; KLEPPE, A. G. *The Object Constraint Language: Getting Your Models Ready for MDA*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. 240 p. ISBN 0321179366.

41 ATLAS Model Management Architecture (AMMA). *Interactive OCL Tutorial*. Available from Internet: <http://atlanmod.emn.fr/atldemo/oclturorial>. Cited 12/24/2009.

42 PIERS, W.; FORTIN, T. *ATL/User Guide - The ATL Language*. Available from Internet: <http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language>. Cited 05/20/2010.

43 JOUAULT, F.; BÉZIVIN, J. KM3: A DSL for metamodel specification. In *Formal Methods for Open Object-Based Distributed Systems*. Bologna, Italy: Springer Berlin / Heidelberg, 2006. (Lecture Notes in Computer Science (LNCS), vol. 4037/2006), p. 171–185. ISBN 978-3-540-34893-1. ISSN 0302-9743 (Print) 1611-3349 (Online). Available from Internet: <http://dx.doi.org/10.1007/11768869>. Cited 10/04/2009.

44 JOUAULT, F.; KURTEV, I. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, NfC, MDD, WUsCAM*. Montego Bay, Jamaica: Springer Berlin / Heidelberg, 2006. (Lecture Notes in Computer Science (LNCS), vol. 3844/2006), p. 128–138. ISBN 978-3-540-31780-7. ISSN 0302-9743 (Print) 1611-3349 (Online). Available from Internet: <http://doi.acm.org/10.1007/11663430>. Cited 10/04/2009.

45 ATLAS Transformation Language (ATL) project. Available from Internet: <http://www-.eclipse.org/m2m/atl>. Cited 08/17/2009.

46 RUSCIO, D. D.; JOUAULT, F.; KURTEV, I.; BÉZIVIN, J.; PIERANTONIO, A. *Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs*. LINA, Université de Nantes - 2, rue de la Houssinière - BP 92208 - 44322 NANTES CEDEX 3, April 2006. 20 p. Available from Internet: <http://hal.archives-ouvertes.fr/docs/00/06/61/21/PDF/rr0602.pdf>. Cited 10/04/2009.

47 The Eclipse Foundation. *The Eclipse Modeling Framework (EMF) Overview*. Available from Internet: <http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc-/references/overview/EMF.html>. Cited 10/05/2009.

48 GEMMI, T. Available from Internet: <http://www.kriha.de/krihaorg/dload/uni-/generativecomputing/generation/CodeGenOverview.jpg>. Cited 10/05/2009.

49 The Eclipse Foundation. *Implementing Model Integrity in EMF with MDT OCL*. Available from Internet: <http://www.eclipse.org/articles/article.php?file=Article-EMF-Codegen-with-OCL/index.html>. Cited 09/04/2009.

50 The Eclipse Foundation. *Generating an EMF Model*. Available from Internet: <http://help-.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/tutorials/clibmod/clibmod.html>. Cited 09/04/2009.

51 The Eclipse Foundation. *The EMF.Edit Framework Overview*. Available from Internet: <http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/references/overview-/EMF.Edit.html>. Cited 09/04/2009.

52 The Eclipse Foundation. *GMF Tutorial*. Available from Internet: <http://wiki.eclipse.org-/GMF_Tutorial>. Cited 09/01/2009.

53 SPIVEY, J. M. *The Z notation: a reference manual*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989. ISBN 0-13-983768-X.

54 JACKSON, D. Nitpick: A checkable specification language. In *In Proceedings of the First ACM SIGSOFT Workshop on Formal Methods in Software Practice*. San Diego, CA, USA: [s.n.], 1996. p. 60–69.

55 GUIZZARDI, G. Modal aspects of object types and part-whole relations and the de re/de dicto distinction. In ____. *Advanced Information Systems Engineering*. Springer Berlin / Heidelberg, 2007. (Lecture Notes in Computer Science (LNCS), vol. 4495/2007), p. 5–20. ISBN 978-3-540-72987-7. Available from Internet: <http://dx.doi.org/10.1007/978-3-540-72988-4_2>. Cited 08/17/2009.

56 GUIZZARDI, G.; WAGNER, G.; SINDEREN, M. van. A formal theory of conceptual modeling universals. In BÜCHEL, G.; KLEIN, B.; ROTH-BERGHOFER, T. (Ed.). *Workshop on Philosophy and Informatics (WSPI), Cologne, Germany, 2004*. Deutsches Forchungszentrum fur Kunstliche Intelligenz, 2004. (CEUR Workshop Proceedings, vol. 112). Available from Internet: <http://doc.utwente.nl/49866/>. Cited 01/24/2010.

57 WHITEHEAD, A. N.; RUSSELL, B. *Principia Mathematica*. Fetter Lane, E.C., London, England: Cambridge University Press, 1910. Available from Internet: <http://name.umdl.umich.edu/AAT3201.0001.001>. Cited 10/04/2009.

58 SIMONS, P. M. *Parts. An Essay in Ontology*. Oxford: Clarendon Press, 1987.

59 BENEVIDES, A. B. *OntoUML Editor Site*. Available from Internet: <http://code.google-.com/p/ontouml>. Cited 09/08/2009.

60 STADNIK, D. *Bug 138179 - Allow to define labels based on attributes of referenced objects*. Available from Internet: <http://bugs.eclipse.org/bugs/show%5Fbug.cgi?id=138179>. Cited 09/08/2009.

61 Free Software Foundation (FSF). *GNU General Public License Version 3 (GPLv3)*. Available from Internet: <http://www.fsf.org/licensing/licenses/gpl.html>. Cited 08/31/2009.

62 The Eclipse Foundation. *Eclipse Public License - v 1.0 (EPLv1.0)*. Available from Internet: <http://www.eclipse.org/org/documents/epl-v10.html>. Cited 08/31/2009.

63 Wikimedia Foundation, Inc. *Copyleft*. Available from Internet: <http://en.wikipedia.org-/wiki/Copyleft>. Cited 01/20/2010.

64 The Eclipse Foundation. *Eclipse Public License (EPL) Frequently Asked Questions*. Available from Internet: <http://www.eclipse.org/legal/eplfaq.php>. Cited 08/31/2009.

65 The Eclipse Foundation. *Model Development Tools (MDT)*. Available from Internet: <http://www.eclipse.org/modeling/mdt>. Cited 08/17/2009.

66 RAYSIDE, D.; CHANG, F.; DENNIS, G.; SEATER, R.; JACKSON, D. Automatic visualization of relational logic models. In MARGARIA, T.; PADBERG, J.; TAENTZER, G.; FISH, A.; KNAPP, A.; STORRLE, H. (Ed.). *Proceedings of the Workshop on the Layout of (Software) Engineering Diagrams (LED 2007)*. Electronic Communications of the EASST (ECEASST), 2007. Volume X (2007). ISSN 1863-2122. Available from Internet: <http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/94/89>. Cited 09/21/2010.

67 BOWEN, J. P.; BUTLER, R. W.; DILL, D. L.; GLASS, R. L.; GRIES, D.; HALL, A.; HINCHEY, M. G.; HOLLOWAY, C. M.; JACKSON, D.; JONES, C. B.; LUTZ, M. J.; PARNAS, D. L.; RUSHBY, J. M.; WING, J. M.; ZAVE, P. An invitation to formal methods. *IEEE Computer*, vol. 29, no. 4, p. 16–30, 1996.

68 XU, F.; CAREY, S. Infants' metaphysics: The case of numerical identity. *Cognitive Psychology*, vol. 30, no. 0005, p. 111–153, 1996. Available from Internet: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.59.1268\&rep=rep1\&type=pdf>. Cited 05/03/2010.

69 BENEVIDES, A. B.; GUIZZARDI, G. A model-based tool for conceptual modeling and domain ontology engineering in OntoUML. In FILIPE, J.; CORDEIRO, J. (Ed.). *ICEIS*. Heidelberg: Springer, 2009. (Lecture Notes in Business Information Processing, vol. 24), p. 528–538. ISBN 978-3-642-01346-1. Available from Internet: <http://www.inf.ufes.br/˜gguizzardi/ICEIS_2009.pdf>. Cited 01/27/2010.

70 BENEVIDES, A. B.; GUIZZARDI, G.; BRAGA, B. F. B.; ALMEIDA, J. P. A. Assessing modal aspects of ontouml conceptual models in alloy. In HEUSER, C. A.; PERNUL, G. (Ed.). *Proceedings of the first International Workshop on Evolving Theories of Conceptual Modelling (ETheCoM 2009), 28th International Conference on Conceptual Modeling (ER 2009)*. Gramado, Brazil: Springer, 2009. (Lecture Notes in Computer Science (LNCS), vol. 5833), p. 55–64. ISBN 978-3-642-04946-0. Available from Internet: <http://www.inf.ufes.br/˜gguizzardi/benevides-et-al-2009.pdf>. Cited 01/27/2010.

71 BENEVIDES, A. B.; GUIZZARDI, G.; BRAGA, B. F. B.; ALMEIDA, J. P. A. Assessing modal aspects of OntoUML conceptual models in alloy. *Journal of Universal Computer Science (J.UCS) Special Issue on Evolving Theories of Conceptual Modelling*, 2011. (Forthcoming).

72 BRAGA, B. F. B.; ALMEIDA, J. P. A.; GUIZZARDI, G.; BENEVIDES, A. B. Transforming OntoUML into Alloy: Towards conceptual model validation using a lightweight formal method. In *2nd IEEE International Workshop UML and Formal Methods (UML&FM 2009) at the 11th International Conference on Formal Engineering Methods (ICFEM 2009)*. Rio de Janeiro: [s.n.], 2009.

73 BRAGA, B. F. B.; ALMEIDA, J. P. A.; GUIZZARDI, G.; BENEVIDES, A. B. Transforming OntoUML into Alloy: Towards conceptual model validation using a lightweight formal method. *Innovations in Systems and Software Engineering (ISSE)*, Springer-Verlag, London, vol. 6, no. 1, p. 55–63, 2010. (Print). Available from Internet: <http://www.springerlink.com/content/m1715n1220717l58/fulltext.pdf>. Cited 09/21/2010.

74 GUARINO, N.; WELTY, C. Evaluating ontological decisions with ontoclean. *Communications of the ACM*, vol. 45, no. 2, p. 61–65, February 2002. Available from Internet: <http://dx.doi.org/10.1145/503124.503150>. Cited 01/13/2010.

75 Laboratory for Applied Ontology (LOA). *OntoClean*. Available from Internet: <http://www.ontoclean.org>. Cited 08/20/2009.

76 Stanford Center for Biomedical Informatics Research. *Protégé OWL Editor*. Available from Internet: <http://protege.stanford.edu>. Cited 08/20/2009.

77 Sandpiper Software. *Visual Ontology Modeler*. Available from Internet: <http://www-.sandsoft.com>. Cited 08/20/2009.

78 PEASE, A. *Sigma*. Available from Internet: <http://sigmakee.sourceforge.net>. Cited 08/20/2009.

79 BEATO, M. E.; BARRIO-SOLÓRZANO, M.; CUESTA, C. E. UML automatic verification tool (TABU). In *SAVCBS'04 Specification and Verification of Component-Based Systems at ACM SIGSOFT 2004/FSE-12*. [S.l.: s.n.], 2004.

80 SCHINZ, I.; TOBEN, T.; MRUGALLA, C.; WESTPHAL, B. The Rhapsody UML verification environment. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference*. [S.l.]: IEEE Computer Society, 2004. p. 174–183. ISBN 0-7695-2222-X.

81 MASSONI, T.; GHEYI, R.; BORBA, P. A UML class diagram analyzer. In *3rd International Workshop on Critical Systems Development with UML (CSDUML'04), affiliated with 7th «UML» Conference*. Lisbon: Springer-Verlag, 2004. p. 143–153.

82 ANASTASAKIS, K.; BORDBAR, B.; GEORG, G.; RAY, I. UML2Alloy: A challenging model transformation. In ENGELS, G.; OPDYKE, B.; SCHMIDT, D.; WEIL, F. (Ed.). *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*. Nashville, USA: Springer, 2007. (LNCS, vol. 4735), p. 436–450. Available from Internet: <http://kyriakos.anastasakis.net/prof/pubs/models07.pdf>. Cited 09/24/2010.

83 VISSERS, C.; SINDEREN, M. van; PIRES, L. F. What makes industries believe in formal methods. In *Proceedings of the 13th International Symposium on Protocol Specification, Testing, and Verification (PSTV XIII)*. Amsterdam: Elsevier Science Publishers, 1993. p. 3–26.

84 ARTALE, A.; GUARINO, N.; KEET, C. M. Formalising temporal constraints on part-whole relations. In BREWKA, G.; LANG, J. (Ed.). *KR*. [S.l.]: AAAI Press, 2008. p. 673–683. ISBN 978-1-57735-384-3.

85 GUIZZARDI, G.; HALPIN, T. Ontological foundations for conceptual modeling. *Journal of Applied Ontology*, vol. 3, no. 1-2, p. 91–110, 2008. ISSN 1570-5838. Available from Internet: <http://dx.doi.org/10.3233/AO-2008-0049>. Cited 08/20/2009.

86 BENEVIDES, A. B. *How to install and run OntoUML Editor*. Available from Internet: <http://code.google.com/p/ontouml/wiki/How_to_install_and_run_OntoUML_Editor>. Cited 09/01/2009.

87 BENEVIDES, A. B. *How to use Cut/Copy/Paste in OntoUML Editor*. Available from Internet: <http://code.google.com/p/ontouml/wiki-/How_to_use_Cut_Copy_Paste_in_OntoUML_Editor>. Cited 09/01/2009.

88 BENEVIDES, A. B. *How to install and run the OntoUML2Alloy ATL transformation*. Available from Internet: <http://code.google.com/p/ontouml/wiki-/How_to_install_and_run_the_OntoUML2Alloy_ATL_transformation>. Cited 11/03/2009.

In the following appendices, one can find the complete set of OCL formalizations for the syntactical constraints shown in the OntoUML profile (11, pp. 317–320, 334–338, 348–352) (appendix A); the ATL transformation that automatically transforms OntoUML models into Alloy specifications (appendix B); the theme customizations for the Alloy Analyzer tool, created in order to customize the visualization of the generated instances (appendix E); some manuals of how to install the OntoUML Graphical Editor in Eclipse (appendix C) and use the OntoUML to Alloy ATL transformation (appendix D). In the annexes, we show the Free and Open Source Software (FOSS) licenses that we have utilized to license our software (annexes A and B).

APPENDIX A – Definition of the OntoUML Syntactical Constraints in OCL

This chapter documents the set of OCL specifications created in order to formalize the OntoUML syntactical constraints. Therefore, we will revisit the original constraints shown in OntoUML profile shown in 11, pp. 317–320, 334–338, 348–352 by adding the OCL specification for every syntactical constraint elaborated in the profile.

So, beneath, we revisit the first part of the OntoUML profile.

A.1 Definition of the Original OntoUML Invariants in OCL

OntoUML profile regarding the categories depicted in Fig. 9

Metaclass: Substance Sortal

Description: *Substance Sortal* is an abstract metaclass that represents the general properties of all *substance sortals*, *i.e.*, rigid, relationally independent object universals that supply a principle of identity for their instances. Substance Sortal has no concrete syntax. Thus, symbolic representations are defined by each of its concrete subclasses.

Constraints:

1. Every substantial object represented in a conceptual model using this profile must be an instance of a substance sortal, directly or indirectly. This means that every concrete

element of this profile used in a class diagram (isAbstract = false) must include in its general collection one class stereotyped as either «kind», «quantity» or «collective»;

- Formalized as an OCL invariant verified in batch verification[1]:

* This invariant uses the EOperation allSuperTypes(), which is specified in OCL in the section A.3.

Listing 18: OCL expression for the first constraint on the metaclass Substance Sortal.

```
context ObjectClass
inv SubstanceSortalConstraint1: if ((self.isAbstract = false) and not self.oclIsKindOf(SubstanceSortal)) then self.allSuperTypes()->exists(x | x.oclIsKindOf(SubstanceSortal)) else true endif
```

2. A substantial object represented in a conceptual model using this profile cannot be an instance of more than one ultimate substance sortal. This means that any stereotyped class in this profile used in a class diagram must not include in its general collection more than one substance sortal class. Moreover, a substance sortal must also not include another substance sortal nor a «subkind» in its general collection, *i.e.*, a substance sortal cannot have as a supertype a member of {«kind», «subkind», «quantity», «collective»};

- Formalized as two OCL invariants verified in live verification[2]:

* These invariants uses the EOperations allSubTypes() and allSuperTypes(), which are specified in OCL in the section A.3.

Listing 19: OCL expressions for the second constraint on the metaclass Substance Sortal.

```
context Generalization
inv SubstanceSortalConstraint2a: self.specific.allSubTypes()->including(self.specific)->forAll(x | x.allSuperTypes()->select(y | y.oclIsKindOf(SubstanceSortal))->size() <= 1)
context Generalization
```

[1] For an explanation of the batch verification mode, see subsection 4.3.1.

[2] For an explanation of the live verification mode, see subsection 4.3.1.

```
inv SubstanceSortalConstraint2b: if self.specific.
    oclIsKindOf(SubstanceSortal) then not self.
    general.oclIsKindOf(RigidSortalClass) else true
     endif
```

3. A Class representing a rigid substantial universal cannot be a subclass of a Class representing an anti-rigid universal. Thus, a substance sortal cannot have as a supertype (must not include in its general collection) a member of {«phase», «role», «roleMixin»}.

   - Formalized as an OCL invariant verified in live verification:

     Listing 20: OCL expression for the third constraint on the metaclass Substance Sortal.

```
context Generalization
inv SubstanceSortalConstraint3: if self.specific.
    oclIsKindOf(SubstanceSortal) then not (self.
    general.oclIsKindOf(AntiRigidSortalClass) or
    self.general.oclIsKindOf(RoleMixin)) else true
    endif
```

Stereotype: «collective»

Description: A «collective» represents a substance sortal whose instances are *collectives*, *i.e.*, they are collections of complexes that have a uniform structure. Examples include a deck of cards, a forest, a group of people, a pile of bricks. Collectives can typically relate to complexes via a constitution relation. For example, a pile of bricks that *constitutes* a wall, a group of people that *constitutes* a football team. In this case, the collectives typically have an extensional principle of identity, in contrast to the complexes they constitute. For instance, The Beatles was in a given world w constituted by the collective {John, Paul, George, Pete} and in another world w' constituted by the collective {John, Paul, George, Ringo}. The replacement of Pete Best by Ringo Star does not alter the identity of the *band*, but creates a numerically different *group of people*.

Constraints:

1. A collective can be extensional. In this case the meta-attribute *isExtensional* is equal to *True*. This means that all its parts are essential and the change (or destruction) of any of its parts terminates the existence of the collective. We use the symbol {extensional} to represent an extensional collective.

   - Formalized as an OCL invariant verified in live verification:

     Listing 21: OCL expression for the first constraint on the stereotype «collective».

     ```
     context Meronymic
     inv CollectiveConstraint1: (not self.isEssential)
         implies self.source->forAll(x | if x.
         oclIsKindOf(Property) then (if x.oclAsType(
         Property).endType.oclIsKindOf(Collective) then
         not x.oclAsType(Property).endType.oclAsType(
         Collective).isExtensional else true endif) else
          false endif)
     ```

Stereotype: «subkind»

Description: A «subkind» is a rigid, relationally independent restriction of a substance sortal that carries the principle of identity supplied by it. An example could be the subkind MalePerson of the kind Person. In general, the stereotype «subkind» can be omitted in conceptual models without loss of clarity.

Constraints:

1. A «subkind» cannot have as a supertype (must not include in its general collection) a member of {«phase», «role», «roleMixin»}.

   - Formalized as an OCL invariant verified in live verification:

     Listing 22: OCL expression for the first constraint on the stereotype «subkind».

     ```
     context Generalization
     inv SubKindConstraint1: if self.specific.
         oclIsKindOf(SubKind) then not (self.general.
     ```

```
    oclIsKindOf(AntiRigidSortalClass) or self.
    general.oclIsKindOf(RoleMixin)) else true endif
```

Stereotype: «phase»

Description: A «phase» represents the phased-sortals phase, *i.e.* anti-rigid and relationally independent universals defined as part of a partition of a substance sortal. For instance, ⟨Caterpillar, Butterfly⟩ partitions the kind Lepdopterum.

Constraints:

1. Phases are anti-rigid universals and, thus, a «phase» cannot appear in a conceptual model as a supertype of a rigid universal;

   - Formalized as an OCL invariant verified in live verification. A wider constraint is implemented by the union of CategoryConstraint1, SubkindConstraint1 and SubstanceSortalConstraint3, so this constraint is not needed:

     Listing 23: OCL expression for the first constraint on the stereotype «phase».

     ```
     context Phase
     inv PhaseConstraint1: if (self.specific.oclIsKindOf
         (RigidSortalClass) or self.specific.oclIsKindOf
         (Category)) then not self.general.oclIsKindOf(
         Phase) else true endif
     ```

2. The phases $\{P_1 \ldots P_n\}$ that form a phase-partition of a substance sortal S are represented in a class diagram as a disjoint and complete generalization set. In other words, a GeneralizationSet with (isCovering = true) and (isDisjoint = true) is used in a representation mapping as the representation for the ontological concept of a phase-partition.

   - Formalized as an OCL invariant verified in batch verification:

     Listing 24: OCL expression for the second constraint on the stereotype «phase».

     ```
     context Phase
     ```

```
inv PhaseConstraint2: let general_substance_sortal
    : SubstanceSortal = Generalization.allInstances
    ()->select(x | (x.specific = self) and (x.
    general.oclIsKindOf(SubstanceSortal)))->collect
    (x | x.general.oclAsType(SubstanceSortal))->any
    (true) in (let phase_generalizations : Set(
    Generalization) = Generalization.allInstances()
    ->select(x | (x.general =
    general_substance_sortal) and (x.specific.
    oclIsKindOf(Phase))) in (let
    phase_generalization_sets : Set(
    GeneralizationSet) = GeneralizationSet.
    allInstances()->select(x | x.generalization->
    includesAll(phase_generalizations)) in (if (
    general_substance_sortal.oclIsUndefined() or (
    phase_generalizations->size() = 1)) then true
    else ((phase_generalization_sets->size() = 1)
    and (phase_generalization_sets->forAll(x | (x.
    isCovering = true) and (x.isDisjoint = true))))
     endif)))
```

Stereotype: «role»

Description: A «role» represents a phased-sortal role, *i.e.* anti-rigid and relationally dependent universal. For instance, the role student is played by an instance of the kind Person.

Constraints:

1. Roles are anti-rigid universals and, thus, a «role» cannot appear in a conceptual model as a supertype of a rigid universal.

    - Formalized as an OCL invariant verified in live verification. A wider constraint is implemented by the union of CategoryConstraint1, SubkindConstraint1 and SubstanceSortalConstraint3, so this constraint is not needed:

Listing 25: OCL expression for the first constraint on the stereotype «role».

```
context Role
inv RoleConstraint1: if (self.specific.oclIsKindOf(
    RigidSortalClass) or self.specific.oclIsKindOf(
    Category)) then not self.general.oclIsKindOf(
    Role) else true endif
```

Metaclass: Mixin Class

Description: *Mixin Class* is an abstract metaclass that represents the general properties of all *mixins*, *i.e.*, non-sortals (or dispersive universals). Mixin Class has no concrete syntax. Thus, symbolic representations are defined by each of its concrete subclasses.

Constraints:

1. A class representing a non-sortal universal cannot be a subclass of a class representing a Sortal. As a consequence of this postulate we have that a mixin class cannot have as a supertype (must not include in its general collection) a member of {«kind», «quantity», «collective», «subkind», «phase», «role»};

   - Formalized as an OCL invariant verified in live verification:

   Listing 26: OCL expression for the first constraint on the metaclass Mixin Class.

```
context Generalization
inv MixinClassConstraint1: if self.specific.
    oclIsKindOf(MixinClass) then not self.general.
    oclIsKindOf(SortalClass) else true endif
```

2. A non-sortal cannot have direct instances. Therefore, a mixin class must always be depicted as an abstract class (isAbstract = true).

   - Formalized as an OCL invariant verified in live verification:

   Listing 27: OCL expression for the second constraint on the metaclass Mixin Class.

```
context MixinClass
inv MixinClassConstraint2: self.isAbstract = true
```

Stereotype: «category»

Description: A «category» represents a rigid and relationally independent *mixin*, *i.e.*, a dispersive universal that aggregates essential properties which are common to different substance sortals. For example, the category RationalEntity as a generalization of Person and IntelligentAgent.

Constraints:

1. A «category» cannot have a «roleMixin» as a supertype. In other words, together with condition 1 for all mixins we have that a «category» can only be subsumed by another «category» or a «mixin».

   - Formalized as an OCL invariant verified in live verification:

     Listing 28: OCL expression for the first constraint on the stereotype «category».

     ```
     context Generalization
     inv CategoryConstraint1: if self.specific.
         oclIsKindOf(Category) then (self.general.
         oclIsKindOf(Category) or self.general.
         oclIsKindOf(Mixin)) else true endif
     ```

Stereotype: «mixin»

Description: A «mixin» represents properties which are essential to some of its instances and accidental to others (semi-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.

Constraints:

1. A «mixin» cannot have a «roleMixin» as a supertype.

   - Formalized as an OCL invariant verified in live verification:

Listing 29: OCL expression for the first constraint on the stereotype «mixin».

```
context Generalization
inv MixinConstraint1: if self.specific.oclIsKindOf(
    Mixin) then not self.general.oclIsKindOf(
    RoleMixin) else true endif
```

Now, we revisit the second part of the OntoUML profile.

OntoUML profile regarding the categories depicted in Fig. 15

Stereotype: «role»

Constraints:

2. Every «role» class must be connected to an association end of a «mediation» relation.

   - Formalized as an OCL invariant verified in batch verification:

   * This invariant uses the EOperation allSuperTypes(), which is specified in OCL in the section A.3.

Listing 30: OCL expression for the second constraint on the stereotype «role».

```
context Role
inv RoleConstraint2: Mediation.allInstances()->
    exists(x | x.target->exists(y | if y.
    oclIsKindOf(Property) then ((y.oclAsType(
    Property).endType = self) or (self.
    allSuperTypes()->includes(y.oclAsType(Property)
    .endType))) else false endif))
```

Stereotype: «roleMixin»

Description: A «roleMixin» represents an anti-rigid and externally dependent non-sortal, *i.e.*, a dispersive universal that aggregates properties which are common to different roles. In includes formal roles such as whole and part, and initiator and responder.

Constraints:

1. Every «roleMixin» class must be connected to an association end of a «mediation» relation.

   - Formalized as an OCL invariant verified in batch verification:

     Listing 31: OCL expression for the first constraint on the stereotype «roleMixin».

     ```
     context RoleMixin
     inv RoleMixinConstraint1: Mediation.allInstances()
         ->exists(x | x.target->exists(y | if y.
         oclIsKindOf(Property) then (y.oclAsType(
         Property).endType = self) else false endif))
     ```

Stereotype: «mode»

Description: A «mode» universal is an intrinsic moment universal. Every instance of mode universal is existentially dependent of exactly one entity. Examples include skills, thoughts, beliefs, intentions, symptoms, private goals.

Constraints:

1. Every «mode» must be (directly or indirectly) connected to an association end of at least one «characterization» relation.

   - Formalized as an OCL invariant verified in batch verification:

   * This invariant uses the EOperations isConected(x:Element) and allSuperTypes(), which are specified in OCL in the section A.3.

Listing 32: OCL expression for the first constraint on the stereotype «mode».

```
context Mode
inv ModeConstraint1: Characterization.allInstances
    ()->exists(x | x.isConected(self) or self.
    allSuperTypes()->exists(y | x.isConected(y)))
```

Stereotype: «relator»

Description: A «relator» universal is a relational moment universal. Every instance of relator universal is existentially dependent of at least two distinct entities. Relators are the instantiation of relational properties such as marriages, kisses, handshakes, commitments, and purchases.

Constraints:

1. Every «relator» must be (directly or indirectly) connected to an association end on at least one «mediation» relation;

   - Formalized as an OCL invariant verified in batch verification:

   * This invariant uses the EOperations isConected(x:Element) and allSuperTypes(), which are specified in OCL in the section A.3.

   Listing 33: OCL expression for the first constraint on the stereotype «relator».

   ```
   context Relator
   inv RelatorConstraint1: Mediation.allInstances()->
       exists(x | x.isConected(self) or self.
       allSuperTypes()->exists(y | x.isConected(y)))
   ```

2. Let R be a relator universal and let $\{C_1 \ldots C_n\}$ be a set of universals mediated by R (related to R via a «mediation» relation). Finally, let lower_{C_i} be the value of the minimum cardinality constraint of the association end connected to C_i in the «mediation» relation. Then, we have that $(\sum_{i=1}^{n} \text{lower}_{C_i}) \geq 2$.

   - Formalized as an OCL invariant verified in batch verification:

* This invariant uses the EOperation allSuperTypes(), which is specified in OCL in the section A.3.

Listing 34: OCL expression for the second constraint on the stereotype «relator».

```
context Relator
inv RelatorConstraint2: Mediation.allInstances()->
    select(x | x.source->exists(y | if y.
    oclIsKindOf(Property) then ((y.oclAsType(
    Property).endType = self) or self.allSuperTypes
    ()->includes(y.oclAsType(Property).endType))
    else false endif))->collect(z | z.target->
    collect(w | if w.oclIsKindOf(Property) then (if
     (w.oclAsType(Property).lower = -1) then 2 else
     w.oclAsType(Property).lower endif) else 0
    endif)->sum())->sum() >= 2
```

Stereotype: «mediation»

Description: A «mediation» is a formal relation that takes place between a relator universal and the endurant universal(s) it mediates. For example, the universal Marriage mediates the role universals Husband and Wife, the universal Enrollment mediates Student and University, and the universal Covalent Bond mediates the universal Atom.

Constraints:

1. An association stereotyped as «mediation» must have in its source association end a class stereotyped as «relator» representing the corresponding relator universal (self.source. oclIsTypeOf(Relator)=true);

    - Formalized as an OCL invariant verified in live verification:

Listing 35: OCL expression for the first constraint on the stereotype «mediation».

```
context Mediation
inv MediationConstraint1: self.source->forAll(x |
    if x.oclIsKindOf(Property) then x.oclAsType(
    Property).endType.oclIsTypeOf(Relator) else
    false endif)
```

2. The association end connected to the mediated universal must have the minimum cardinality constraints of at least one (self.target.lower ≥ 1);

   - Formalized as an OCL invariant verified in live verification:

   Listing 36: OCL expression for the second constraint on the stereotype «mediation».

```
context Property
inv MediationConstraint2: if self.target.
    oclIsKindOf(Mediation) then ((self.lower >= 1)
    or (self.lower = -1)) else true endif
```

3. The association end connected to the mediated universal must have the property (self.target.isReadOnly = true);

   - Formalized as an OCL invariant verified in live verification:

   Listing 37: OCL expression for the third constraint on the stereotype «mediation».

```
context Property
inv MediationConstraint3: if self.target.
    oclIsKindOf(Mediation) then self.isReadOnly
    else true endif
```

4. The association end connected to the relator universal must have the minimum cardinality constraints of at least one (self.source.lower ≥ 1);

   - Formalized as an OCL invariant verified in live verification:

   Listing 38: OCL expression for the fourth constraint on the stereotype «mediation».

```
context Property
inv MediationConstraint4: if self.source.
    oclIsKindOf(Mediation) then ((self.lower >= 1)
    or (self.lower = -1)) else true endif
```

5. «mediation» associations are always binary associations.

   - Formalized as an OCL invariant verified in batch verification:

     Listing 39: OCL expression for the fifth constraint on the stereotype «mediation».

     ```
     context Mediation
     inv MediationConstraint5: self.relatedElement->size
         () = 2
     ```

Stereotype: «characterization»

Description: A «characterization» is a formal relation that takes place between a mode universal and the endurant universal this mode universal characterizes. For example, the universals Private Goal and Capability characterize the universal Agent.

Constraints:

1. An association stereotyped as «characterization» must have in its source association end a class stereotyped as «mode» representing the characterizing mode universal (self.source.oclIsTypeOf(Mode)=true);

   - Formalized as an OCL invariant verified in live verification:

     Listing 40: OCL expression for the first constraint on the stereotype «characterization».

     ```
     context Characterization
     inv CharacterizationConstraint1: self.source->
         forAll(x | if x.oclIsKindOf(Property) then x.
         oclAsType(Property).endType.oclIsTypeOf(Mode)
         else false endif)
     ```

2. The association end connected to the characterized universal must have the cardinality constraints of one and exactly one (self.target.lower = 1 and self.target.upper = 1);

   - Formalized as an OCL invariant verified in live verification:

Listing 41: OCL expression for the second constraint on the stereotype «characterization».

```
context Property
inv CharacterizationConstraint2: if self.target.
      oclIsKindOf(Characterization) then ((self.lower
       = 1) and (self.upper = 1)) else true endif
```

3. The association end connected to the characterizing quality universal (source association end) must have the minimum cardinality constraints of one (self.source.lower $\geq$ 1);

    - Formalized as an OCL invariant verified in live verification:

      Listing 42: OCL expression for the third constraint on the stereotype «characterization».

```
context Property
inv CharacterizationConstraint3: if self.source.
      oclIsKindOf(Characterization) then ((self.lower
       >= 1) or (self.lower = -1)) else true endif
```

4. The association end connected to the characterized universal must have the property (self.target.isReadOnly = true);

    - Formalized as an OCL invariant verified in live verification:

      Listing 43: OCL expression for the fourth constraint on the stereotype «characterization».

```
context Property
inv CharacterizationConstraint4: if self.target.
      oclIsKindOf(Characterization) then self.
      isReadOnly else true endif
```

5. «characterization» associations are always binary associations.

    - Formalized as an OCL invariant verified in batch verification:

Listing 44: OCL expression for the fifth constraint on the stereotype «characterization».

```
context Characterization
inv CharacterizationConstraint5: self.
    relatedElement->size() = 2
```

Stereotype: Derivation Relation

Description: A derivation relation represents the formal relation of derivation that takes place between a material relation and the relator universal this material relation is derived from. Examples include the material relation married-to, which is derived from the relator universal Marriage, the material relation kissed-by, derived from the relator universal Kiss, and the material relation purchases-from, derived from the relator universal Purchase.

Constraints:

1. A derivation relation must have one of its association ends connected to a relator universal (the black circle end) and the other one connected to a material relation (self.target.oclIsTypeOf(Relator)=true, self.source.oclIsTypeOf(Material Association)=true);

   - Formalized as two OCL invariants verified in live verification:

     Listing 45: OCL expressions for the first constraint on the stereotype Derivation Relation.

```
context Derivation
inv DerivationConstraint1a: self.source->forAll(x |
     if x.oclIsKindOf(Property) then x.oclAsType(
    Property).endType.oclIsTypeOf(
    MaterialAssociation) else false endif)
context Derivation
inv DerivationConstraint1b: self.target->forAll(x |
     if x.oclIsKindOf(Property) then x.oclAsType(
      Property).endType.oclIsTypeOf(Relator) else
      false endif)
```

2. Derivation associations are always binary associations;

   - Formalized as an OCL invariant verified in batch verification:

     Listing 46: OCL expression for the second constraint on the stereotype Derivation Relation.

     ```
     context Derivation
     inv DerivationConstraint2: self.relatedElement->
           size() = 2
     ```

3. The black circle end of the derivation relation must have the cardinality constraints of one and exactly one (self.target.lower = 1 and self.target.upper = 1);

   - Formalized as an OCL invariant verified in live verification:

     Listing 47: OCL expression for the third constraint on the stereotype Derivation Relation.

     ```
     context Property
     inv DerivationConstraint3: if self.target.
           oclIsKindOf(Derivation) then ((self.lower = 1)
           and (self.upper = 1)) else true endif
     ```

4. The black circle end of the derivation relation must have the property (self.target.isReadOnly = true);

   - Formalized as an OCL invariant verified in live verification:

     Listing 48: OCL expression for the fourth constraint on the stereotype Derivation Relation.

     ```
     context Property
     inv DerivationConstraint4: if self.target.
           oclIsKindOf(Derivation) then self.isReadOnly
           else true endif
     ```

5. The cardinality constraints of the association end connected to the material relation in a derivation relation are a product of the cardinality constraints of the «mediation» relations of the relator universal that this material relation derives from. This is done in the manner previously shown in subsection 4.3.3. However, since «mediation» relations require a minimum cardinality of one on both of its association ends, then the minimum cardinality on the material relation end of a derivation relation must also be ≥ 1 (self.source.lower ≥ 1).

   - Formalized as an OCL invariant verified in live verification:

     Listing 49: OCL expression for the fifth constraint on the stereotype Derivation Relation.

```
context Property
inv DerivationConstraint5: if self.source.
    oclIsKindOf(Derivation) then ((self.lower >= 1)
     or (self.lower = -1)) else true endif
```

Stereotype: «material»

Description: A «material» association represents a material relation, *i.e.*, a relational universal which is induced by a relator universal. Examples include student *studies in* university, patient is *treated in* medical unit, person is *married to* person.

Constraints:

1. Every «material» association must be connected to the association end of exactly one derivation relation;

   - Formalized as an OCL invariant verified in batch verification:

     Listing 50: OCL expression for the first constraint on the stereotype «material».

```
context MaterialAssociation
inv MaterialAssociationConstraint1: Derivation.
    allInstances()->one(x | x.source->exists(y | if
     y.oclIsKindOf(Property) then (y.oclAsType(
    Property).endType = self) else false endif))
```

2. The cardinality constraints of the association ends of a material relation are derived from the cardinality constraints of the «mediation» relations of the relator universal that this material relation is derived from. This is done in the manner shown in subsection 4.3.3. However, since «mediation» relations require a minimum cardinality of one on both of its association ends, then the minimum cardinality constraint on each end of the derived material relation must also be ≥ 1;

    - Formalized as an OCL invariant verified in live verification:

    Listing 51: OCL expression for the second constraint on the stereotype «material».

```
context Property
inv MaterialAssociationConstraint2: if self.
    associationEnd.oclIsKindOf(MaterialAssociation)
     then ((self.lower >= 1) or (self.lower = -1))
    else true endif
```

3. Every «material» association must have the property (isDerived = true).

    - Formalized as an OCL invariant verified in live verification:

    Listing 52: OCL expression for the third constraint on the stereotype «material».

```
context MaterialAssociation
inv MaterialAssociationConstraint3: self.isDerived
```

Metaclass: Property

Description: An attribute in the UML metamodel is a property owned by a classifier. Attributes are used in this profile to represent attribute functions derived for quality universals. Examples are the attributes color, age, and startingDate.

Constraints:

1. A property owned by a classifier (representing an attribute of that classifier) must have the minimum cardinality constraints of one (self.lower $\geq$ 1).

    - Formalized as an OCL invariant verified in live verification:

Listing 53: OCL expression for the first constraint on the metaclass Property.

```
context DatatypeRelationship
inv PropertyConstraint1: self.target->forAll(x | if
     x.oclIsKindOf(Property) then ((x.oclAsType(
    Property).lower >= 1) or (x.oclAsType(Property)
    .lower = -1)) else false endif)
```

Finally, we revisit the last part of the OntoUML profile.

OntoUML profile regarding the categories depicted in Fig. 17

Metaclass: Meronymic

Description: Abstract metaclass representing the general properties of all meronymic relations. Meronymic has no concrete syntax. Thus, symbolic representations are defined by each of its concrete subclasses.

Constraints:

1. Weak supplementation: Let U be a universal whose instances are wholes and let $\{C_1 \ldots C_n\}$ be a set of universals related to U via aggregation relations. Let lower_{C_i} be the value of the minimum cardinality constraint of the association end connected to C_i in the aggregation relation. Then, we have that $(\sum_{i=1}^{n} \text{lower}_{C_i}) \geq 2$;
   - Formalized as an OCL invariant verified in batch verification:

Listing 54: OCL expression for the first constraint on the metaclass Meronymic.

```
context Meronymic
inv MeronymicConstraint1: Meronymic.allInstances()
    ->select(x | x.source->exists(y | if y.
    oclIsKindOf(Property) then (self.source->exists
    (z | if z.oclIsKindOf(Property) then (z.
oclAsType(Property).endType = y.oclAsType(
Property).endType) else false endif)) else
false endif))->collect(w | w.target->collect(k
| if k.oclIsKindOf(Property) then (if (k.
oclAsType(Property).lower = -1) then 2 else k.
oclAsType(Property).lower endif) else 0 endif)
->sum())->sum() >= 2
```

2. Essential Parthood: The *isEssential* attribute represents whether the meronymic relation is one of essential parthood, *i.e.*, whether the part is essential to the whole. In case the classifier connected to the association end representing the whole is an anti-rigid classifier, then the meta-attribute *isEssential* must be false, whereas the meta-attribute *isImmutable* may be true. However, if *isEssential* is true (in case of a rigid classifier with essential parts) then *isImmutable* must also be true. The concrete representation of this meta-property is via the tagged value essential decorating the association;

   * As the meta-attribute *isImmutable* is defined only for parts in 11, p. 286 and 55, pp. 17-18, we modified the OntoUML metamodel in order to create a related meta-attribute regarding the immutability of wholes by replacing the meta-attribute *isImmutable* by the meta-attribute *isImmutablePart* and creating a new meta-attribute *isImmutableWhole* in the metaclass Meronymic. See Definitions 25 and 30 for *isImmutablePart* and *isImmutableWhole*, respectively.

   - This constraint is formalized as two OCL invariants, in which the first is verified in live verification and the last is verified in batch verification:

     Listing 55: OCL expressions for the second constraint on the metaclass Meronymic.

```
context Meronymic
inv MeronymicConstraint2a: (self.source->forAll(x |
     if x.oclIsKindOf(Property) then (x.oclAsType(
    Property).endType.oclIsKindOf(
    AntiRigidSortalClass) or x.oclAsType(Property).
    endType.oclIsKindOf(AntiRigidMixinClass)) else
    false endif)) implies (self.isEssential = false
    )
context Meronymic
```

```
inv MeronymicConstraint2b: self.isEssential implies
    self.isImmutablePart
```

3. The last two constraints in this metaclass are not actually constraints, but explanations of meta-attributes. Therefore, they have not been formalized in OCL.

Metaclass: componentOf

Description: componentOf is a parthood relation between two complexes. Examples include: (a) my hand is part of my arm; (b) a car engine is part of a car; (c) an ALU is part of a CPU; (d) a heart is part of a circulatory system.

Meta-properties: Non-reflexivity, Anti-Symmetry, Non-Transitivity and Weak Supplementation.

Constraints:

1. The classes connected to both association ends of this relation must represent universals whose instances are functional complexes. A universal X is a universal whose instances are functional complexes if it satisfies the following conditions: (i) If X is a sortal universal, then it must be either stereotyped as «kind» or be a subtype of a class stereotyped as «kind»; (ii) Otherwise, if X is a mixin universal, then for all classes Y such that Y is a subtype of X, we have that Y cannot be either stereotyped as «quantity» or «collective», and Y cannot be a subtype of class stereotyped as either «quantity» or «collective».

   - Formalized as two OCL invariants, both verified in live verification:

   * These invariants uses the EOperation hasFunctionalComplexesInstances(), which is specified in OCL in the section A.3.

   Listing 56: OCL expressions for the first constraint on the metaclass componentOf.

```
context componentOf
inv componentOfConstraint1a: self.source->forAll(x
    | if x.oclIsKindOf(Property) then x.oclAsType(
    Property).endType.
    hasFunctionalComplexesInstances() else false
    endif)
```

```
context componentOf
inv componentOfConstraint1b: self.target->forAll(x | if x.oclIsKindOf(Property) then x.oclAsType(Property).endType.hasFunctionalComplexesInstances() else false endif)
```

Metaclass: subQuantityOf

Description: subQuantityOf is a parthood relation between two quantities. Examples include: (a) alcohol is part of Wine; (b) Plasma is part of Blood; (c) Sugar is part of Ice Cream; (d) Milk is part of Cappuccino.

Meta-properties: Non-reflexivity, Anti-Symmetry, Transitivity and Strong Supplementation (Extensional Mereology).

Constraints:

1. This relation is always non-shareable (isShareable = false);

   - Formalized as an OCL invariant verified in live verification:

     Listing 57: OCL expression for the first constraint on the metaclass subQuantityOf.

     ```
     context subQuantityOf
     inv subQuantityOfConstraint1: self.isShareable = false
     ```

2. All entities stereotyped as «quantity» are extensional individuals and, thus, all parthood relations involving quantities are essential parthood relations;

   - Formalized as an OCL invariant verified in live verification:

     Listing 58: OCL expression for the second constraint on the metaclass subQuantityOf.

     ```
     context subQuantityOf
     ```

```
inv subQuantityOfConstraint2a: self.isEssential =
    true
```

3. The maximum cardinality constraint in the association end connected to the part must be one (self.target.upper = 1);

   - Formalized as an OCL invariant verified in live verification:

   Listing 59: OCL expression for the third constraint on the metaclass subQuantityOf.

```
context Property
inv subQuantityOfConstraint3: if self.target.
    oclIsKindOf(subQuantityOf) then self.upper = 1
    else true endif
```

4. The classes connected to both association ends of this relation must represent universals whose instances are quantities. A universal X is a universal whose instances are quantities if it satisfies the following conditions: (i) If X is a sortal universal, then it must be either stereotyped as «quantity» or be a subtype of a class stereotyped as «quantity»; (ii) Otherwise, if X is a mixin universal, then for all classes Y such that Y is a subtype of X, we have that Y cannot be either stereotyped as «kind» or «collective», and Y cannot be a subtype of class stereotyped as either «kind» or «collective».

   - Formalized as two OCL invariants, both verified in live verification:

   * These invariants uses the EOperation hasQuantitiesInstances(), which is specified in OCL in the section A.3.

   Listing 60: OCL expressions for the fourth constraint on the metaclass subQuantityOf.

```
context subQuantityOf
inv subQuantityOfConstraint4a: self.source->forAll(
    x | if x.oclIsKindOf(Property) then x.oclAsType
    (Property).endType.hasQuantitiesInstances()
    else false endif)
inv subQuantityOfConstraint4b: self.target->forAll(
    x | if x.oclIsKindOf(Property) then x.oclAsType
    (Property).endType.hasQuantitiesInstances()
    else false endif)
```

Metaclass: subCollectionOf

Description: subCollectionOf is a parthood relation between two collectives. Examples include: (a) the north part of the Black Forest is part of the Black Forest; (b) The collection of Jokers in a deck of cards is part of that deck of cards; (c) the collection of forks in cutlery set is part of that cutlery set; (d) the collection of male individuals in a crowd is part of that crowd.

Meta-properties: Non-reflexivity, Anti-Symmetry, Transitivity and Weak Supplementation (Minimum Mereology).

Constraints:

1. The classes connected to both association ends of this relation must represent universals whose instances are collectives. A universal X is a universal whose instances are collectives if it satisfies the following conditions: (i) If X is a sortal universal, then it must be either stereotyped as «collective» or be a subtype of a class stereotyped as «collective»; (ii) Otherwise, if X is a mixin universal, then for all classes Y such that Y is a subtype of X, we have that Y cannot be either stereotyped as «kind» or «quantity», and Y cannot be a subtype of class stereotyped as either «kind» or «quantity»;

   - Formalized as two OCL invariants, both verified in live verification:

   * These invariants uses the EOperation hasCollectivesInstances(), which is specified in OCL in the section A.3.

   Listing 61: OCL expressions for the first constraint on the metaclass subCollectionOf.

```
context subCollectionOf
inv subCollectionOfConstraint1a: self.source->
    forAll(x | if x.oclIsKindOf(Property) then x.
    oclAsType(Property).endType.
    hasCollectivesInstances() else false endif)
inv subCollectionOfConstraint1b: self.target->
    forAll(x | if x.oclIsKindOf(Property) then x.
    oclAsType(Property).endType.
    hasCollectivesInstances() else false endif)
```

2. The maximum cardinality constraint in the association end connected to the part must be one (self.target.upper = 1).

   - Formalized as an OCL invariant verified in live verification:

     Listing 62: OCL expression for the second constraint on the metaclass subCollectionOf.

     ```
     context Property
     inv subCollectionOfConstraint2: if self.target.
         oclIsKindOf(subCollectionOf) then self.upper =
         1 else true endif
     ```

Metaclass: memberOf

Description: memberOf is a parthood relation between a complex or a collective (as a part) and a collective (as a whole). Examples include: (a) a tree is part of forest; (b) a card is part of a deck of cards; (c) a fork is part of cutlery set; (d) a club member is part of a club.

Meta-properties: Non-reflexivity, Anti-Symmetry, Intransitivity and Weak Supplementation. Although transitivity does not hold across two memberOf relations, a memberOf relation followed by subCollectionOf is transitive. That is, for all a,b,c, if memberOf(a,b) and memberOf(b,c) then ¬memberOf(a,c), but if memberOf(a,b) and subCollectionOf(b,c) then memberOf(a,c).

Constraints:

1. This relation can only represent essential parthood (isEssential = true) if the object representing the whole on this relation is an extensional (isExtensional = true) individual. In this case, all parthood relations in which this individual participates as a whole are essential parthood relations;

   - Formalized as an OCL invariant verified in live verification:

     Listing 63: OCL expression for the first constraint on the metaclass memberOf.

     ```
     context Collective
     ```

```
inv memberOfConstraint1: (not self.isExtensional)
    implies (memberOf.allInstances()->select(x | x.
    source->forAll(y | if y.oclIsKindOf(Property)
    then (y.oclAsType(Property).endType = self)
    else false endif))->forAll(x | not x.
    isEssential))
```

2. The classifier connected to association end relative to the whole individual must be a universal whose instances are collectives. The classifier connected to the association end relative to the part can be either a universal whose instances are collectives, or a universal whose instances are functional complexes.

    - Formalized as two OCL invariants, both verified in live verification:

    * These invariants uses the EOperations hasCollectivesInstances() and hasFunctionalComplexesInstances(), which are specified in OCL in the section A.3.

    Listing 64: OCL expressions for the second constraint on the metaclass memberOf.

```
context memberOf
inv memberOfConstraint2a: self.source->forAll(x |
    if x.oclIsKindOf(Property) then x.oclAsType(
    Property).endType.hasCollectivesInstances()
    else false endif)
inv memberOfConstraint2b: self.target->forAll(x |
    if x.oclIsKindOf(Property) then (x.oclAsType(
    Property).endType.hasCollectivesInstances() or
    x.oclAsType(Property).endType.
    hasFunctionalComplexesInstances()) else false
    endif)
```

A.2 Definition of Additional Invariants in OCL

This section documents a set of OCL invariants that are not related to the OntoUML syntactical constraints as documented in the OntoUML profile (11, pp. 317–320, 334–338, 348–352). We create these invariants and formalize them in OCL as part of the OntoUML syntactical constraints because we think they are important in the specification of OntoUML models.

Some of these invariants were not specified in the OntoUML profile because they are similar to invariants taken from the UML metamodel. For example, the invariants that state that the interval specified by the minimum and maximum cardinalities must be a non-empty (possibly infinite) set of natural numbers (*i.e.*, the minimum cardinality must be less or equal than the maximum cardinality) are present in the UML metamodel (13, p. 95, constraint 2). As, we implemented the OntoUML metamodel from scratch, we have to specify these UML invariants in OCL.

Additional Invariants

- In all association ends of the associations stereotyped as «formal» or «material», the maximum cardinality constraint (the property upper) must be equal or greater to the lower cardinality constraint (the property lower);
  - Formalized as an OCL invariant verified in batch verification:

    Listing 65: OCL expression for the first additional constraint on the metaclass Association.

    ```
    context Association
    inv AssociationConstraint1*: self.associationEnd->
        forAll(x | if (x.lower <> -1) then ((x.upper >=
         x.lower) or (x.upper = -1)) else (x.upper =
        -1) endif)
    ```

- A relationship stereotyped as «datatypeRelationship» must have in its target association end a class stereotyped as «simpleDatatype» or «structuredDatatype», because attributes represent attribute functions derived for quality universals;
  - Formalized as an OCL invariant verified in live verification:

    Listing 66: OCL expression for the first additional constraint on the stereotype «datatypeRelationship».

    ```
    context DatatypeRelationship
    ```

```
inv DatatypeRelationshipConstraint1*: self.target->
    forAll(x | if x.oclIsKindOf(Property) then (x.
    oclAsType(Property).endType.oclIsKindOf(
    SimpleDatatype) or x.oclAsType(Property).
    endType.oclIsKindOf(StructuralDatatype)) else
    false endif)
```

- A relationship stereotyped as «datatypeRelationship» must have in its source association end an instance of Classifier, excepting instances of «simpleDatatype», because a «simpleDatatype» is a Datatype that has no attributes. Excepting for Generalizations, GeneralizationSets and Properties, all the other OntoUML constructs are Classifiers;
  - Formalized as an OCL invariant verified in live verification:

    Listing 67: OCL expression for the second additional constraint on the stereotype «datatypeRelationship».

```
context DatatypeRelationship
inv DatatypeRelationshipConstraint2*: self.source->
    forAll(x | if x.oclIsKindOf(Property) then (x.
    oclAsType(Property).endType.oclIsKindOf(
    Classifier) and not x.oclAsType(Property).
    endType.oclIsKindOf(SimpleDatatype)) else false
     endif)
```

- Every relationship stereotyped as «datatypeRelationship» that have a «structuredDatatype» in its source association end must have the meta-attribute isReadOnly = true in its target association end;
  - Formalized as an OCL invariant verified in live verification:

    Listing 68: OCL expression for the third additional constraint on the stereotype «datatypeRelationship».

```
context Property
inv DatatypeRelationshipConstraint3*: (self.target.
    oclIsKindOf(DatatypeRelationship)) implies ((
    not self.isReadOnly) implies (self.target.
```

```
      source->forAll(x | if (x.oclIsKindOf(Property))
       then (not x.oclAsType(Property).endType.
      oclIsKindOf(StructuralDatatype)) else false
      endif)))
```

- An association stereotyped as «characterization», «mediation», «componentOf», «memberOf», «subCollectionOf», «subQuantityOf» or a derivation relation must have its source extremity connected to only one element;
  - Formalized as an OCL invariant verified in batch verification:

    Listing 69: OCL expression for the first additional constraint on the metaclass Directed Binary Relationship.

    ```
    context DirectedBinaryRelationship
    inv DirectedBinaryRelationshipConstraint1: self.
          source->size() = 1
    ```

- An association stereotyped as «characterization», «mediation», «componentOf», «memberOf», «subCollectionOf», «subQuantityOf» or a derivation relation must have its target extremity connected to only one element;
  - Formalized as an OCL invariant verified in batch verification:

    Listing 70: OCL expression for the second additional constraint on the metaclass Directed Binary Relationship.

    ```
    context DirectedBinaryRelationship
    inv DirectedBinaryRelationshipConstraint2: self.
          target->size() = 1
    ```

- An association stereotyped as «characterization», «mediation», «componentOf», «memberOf», «subCollectionOf», «subQuantityOf» or a derivation relation must have its source extremity connected to a Property;
  - Formalized as an OCL invariant verified in live verification:

    Listing 71: OCL expression for the third additional constraint on the metaclass Directed Binary Relationship.

```
context DirectedBinaryRelationship
inv DirectedBinaryRelationshipConstraint3*: self.
    source->forAll(x | x.oclIsKindOf(Property))
```

- An association stereotyped as «characterization», «mediation», «componentOf», «memberOf», «subCollectionOf», «subQuantityOf» or a derivation relation must have its target extremity connected to a Property;

  – Formalized as an OCL invariant verified in live verification:

    Listing 72: OCL expression for the fourth additional constraint on the metaclass Directed Binary Relationship.

```
context DirectedBinaryRelationship
inv DirectedBinaryRelationshipConstraint4*: self.
    target->forAll(x | x.oclIsKindOf(Property))
```

- In the source of the relationships stereotyped «characterization», «mediation», «componentOf», «memberOf», «subCollectionOf», «subQuantityOf» or a derivation relation, the maximum cardinality constraint (the property upper) must be equal or greater to the lower cardinality constraint (the property lower);

  – Formalized as an OCL invariant verified in batch verification:

    Listing 73: OCL expression for the first part of the fifth additional constraint on the metaclass Directed Binary Relationship.

```
context DirectedBinaryRelationship
inv DirectedBinaryRelationshipConstraint5a*: self.
    source->forAll(x | if x.oclIsKindOf(Property)
    then (if (x.oclAsType(Property).lower <> -1)
    then ((x.oclAsType(Property).upper >= x.
    oclAsType(Property).lower) or (x.oclAsType(
    Property).upper = -1)) else (x.oclAsType(
    Property).upper = -1) endif) else false endif)
```

- In the target of the relationships stereotyped «characterization», «mediation», «componentOf», «memberOf», «subCollectionOf», «subQuantityOf» or a derivation relation, the maximum cardinality constraint (the property upper) must be equal or greater to the lower cardinality constraint (the property lower);

- Formalized as an OCL invariant verified in batch verification:

  Listing 74: OCL expression for the second part of the fifth additional constraint on the metaclass Directed Binary Relationship.

```
context DirectedBinaryRelationship
inv DirectedBinaryRelationshipConstraint5b*: self.
    target->forAll(x | if x.oclIsKindOf(Property)
    then (if (x.oclAsType(Property).lower <> -1)
    then ((x.oclAsType(Property).upper >= x.
    oclAsType(Property).lower) or (x.oclAsType(
    Property).upper = -1)) else (x.oclAsType(
    Property).upper = -1) endif) else false endif)
```

- A generalization must have its source extremity connected to at maximum one element;
  - Formalized as an OCL invariant verified in live verification:

    Listing 75: OCL expression for the first additional constraint on the metaclass Generalization.

```
context Generalization
inv GeneralizationConstraint1: self.source->size()
    <= 1
```

- A generalization must have its target extremity connected to at maximum one element;
  - Formalized as an OCL invariant verified in live verification:

    Listing 76: OCL expression for the second additional constraint on the metaclass Generalization.

```
context Generalization
inv GeneralizationConstraint2: self.target->size()
    <= 1
```

- An association stereotyped as «material» must be connected to Properties which have its meta-attributes isDerived = true;
  - Formalized as an OCL invariant verified in live verification:

    Listing 77: OCL expression for the first additional constraint on the stereotype «material».

```
context Property
inv MaterialAssociationConstraint4*: if self.
    associationEnd.oclIsKindOf(MaterialAssociation)
     then isDerived else true endif
```

- The maximum cardinalities of an association stereotyped as «material» are calculated automatically;
  - Formalized as an OCL invariant verified in batch verification:

    Listing 78: OCL expression for the second additional constraint on the stereotype «material».

```
context MaterialAssociation
inv MaterialAssociationConstraint5*: (self.
    associationEnd->at(1).oclAsType(Property).upper
     = self.deriveUpperMaterialAssociationExt1())
    and (self.associationEnd->at(2).oclAsType(
    Property).upper = self.
    deriveUpperMaterialAssociationExt2())
```

- The cardinalities on the source extremity of a derivation relationship are calculated automatically;
  - Formalized as an OCL invariant verified in batch verification:

    Listing 79: OCL expression for the first additional constraint on the derivation relationships.

```
context Derivation
```

```
inv DerivationConstraint6*: (self.source->any(true)
    .oclAsType(Property).lower = self.
    deriveLowerDerivation()) and (self.source->any(
    true).oclAsType(Property).upper = self.
    deriveUpperDerivation())
```

- There cannot be two «mediation» relationships x and y having the same ground, *i.e.*, $domain(x) = domain(y)$ and $codomain(x) = codomain(y)$;
  - Formalized as an OCL invariant verified in live verification:

    Listing 80: OCL expression for the first additional constraint on the stereotype «mediation».

```
context Mediation
inv MediationConstraint6*: Mediation.allInstances()
    ->excluding(self)->select(x | x.source->forAll(
    y | if y.oclIsKindOf(Property) then (self.
    source->forAll(z | if z.oclIsKindOf(Property)
    then ((y.oclAsType(Property).endType = z.
    oclAsType(Property).endType) and (y.oclAsType(
    Property).endType = z.oclAsType(Property).
    endType)) else false endif)) else false endif)
    and x.target->forAll(y | if y.oclIsKindOf(
    Property) then (self.target->forAll(z | if z.
    oclIsKindOf(Property) then ((y.oclAsType(
    Property).endType = z.oclAsType(Property).
    endType) and (y.oclAsType(Property).endType = z
    .oclAsType(Property).endType)) else false endif
    )) else false endif))->isEmpty()
```

- In a relationship stereotyped as «componentOf», «memberOf», «subCollectionOf» or «subQuantityOf» (*i.e.*, a Meronymic relationship), if the meta-attribute isImmutablePart = true, then the Properties connected on its target association ends must have the meta-attribute isReadOnly = true;
  - Formalized as an OCL invariant verified in live verification:

Listing 81: OCL expression for the first additional constraint on the metaclass Meronymic.

```
context Property
inv MeronymicConstraint3*: (self.isReadOnly = false
    ) implies (self.target.oclIsKindOf(Meronymic)
    implies (self.target.oclAsType(Meronymic).
    isImmutablePart = false))
```

- In a relationship stereotyped as «componentOf», «memberOf», «subCollectionOf» or «subQuantityOf» (*i.e.*, a Meronymic relationship), if the meta-attribute isInseparable = true (in case of an anti-rigid class with inseparable parts), then the meta-attribute isImmutableWhole must also be true;
  - Formalized as an OCL invariant verified in batch verification:

    Listing 82: OCL expression for the second additional constraint on the metaclass Meronymic.

    ```
    context Meronymic
    inv MeronymicConstraint4*: self.isInseparable
        implies self.isImmutableWhole -- Extension of
        the second constraint of the metaclass
        Meronymic.
    ```

- In a relationship stereotyped as «componentOf», «memberOf», «subCollectionOf» or «subQuantityOf» (*i.e.*, a Meronymic relationship), if the meta-attribute isImmutableWhole = true, then the Properties connected on its source association ends must have the meta-attribute isReadOnly = true;
  - Formalized as an OCL invariant verified in live verification:

    Listing 83: OCL expression for the third additional constraint on the metaclass Meronymic.

    ```
    context Property
    inv MeronymicConstraint5*: (self.isReadOnly = false
        ) implies (self.source.oclIsKindOf(Meronymic)
    ```

```
    implies (self.source.oclAsType(Meronymic).
    isImmutableWhole = false))
```

- Non-shareability implies a cardinality of exactly one in the extremity connected to the whole (see Definition 20);
  - Formalized as an OCL invariant verified in live verification:

    Listing 84: OCL expression for the fourth additional constraint on the metaclass Meronymic.

```
context Meronymic
inv MeronymicConstraint6*: (self.isShareable =
    false) implies self.source->forAll(x | if x.
    oclIsKindOf(Property) then ((x.oclAsType(
    Property).lower = 1) and (x.oclAsType(Property)
    .upper = 1)) else false endif)
```

- The minimum cardinality constraint (the meta-attribute lower) must be a natural number (ℕ) or the least cardinal infinite $\aleph_0$, which is represented as *;
  - Formalized as an OCL invariant verified in live verification:

    Listing 85: OCL expression for the first additional constraint on the metaclass Multiplicity Element.

```
context MultiplicityElement
inv MultiplicityElementConstraint1: (self.lower >=
    0) or (self.lower = -1)
```

- The maximum cardinality constraint (the meta-attribute upper) must be a natural number (ℕ) or the least cardinal infinite $\aleph_0$, which is represented as *;
  - Formalized as an OCL invariant verified in live verification:

    Listing 86: OCL expression for the second additional constraint on the metaclass Multiplicity Element.

```
context MultiplicityElement
```

```
inv MultiplicityElementConstraint2: (self.upper >=
    0) or (self.upper = -1)
```

- A class stereotyped as «structuredDatatype» must have at least two disjoint attributes (a class can have attributes by means of relationships stereotyped as «structuredDatatype»);
  - Formalized as an OCL invariant verified in batch verification:

    Listing 87: OCL expression for the first additional constraint on the stereotype «structuredDatatype».

```
context StructuredDatatype
inv StructuredDatatypeConstraint1*:
    DatatypeRelationship.allInstances()->collect(x
    | if x.source->exists(y | if y.oclIsKindOf(
    Property) then (y.oclAsType(Property).endType =
     self) else false endif) then x.target->collect
    (z | if z.oclIsKindOf(Property) then (if (z.
    oclAsType(Property).lower = -1) then 2 else z.
    oclAsType(Property).lower endif) else 0 endif)
    ->sum() else 0 endif)->sum() >= 2
```

- A relationship stereotyped as «subQuantityOf» must have the meta-attribute isImmutablePart = true, because these relationships are always essential (*i.e.*, the meta-attribute isEssential = true);
  - Formalized as an OCL invariant verified in live verification:

    Listing 88: OCL expression for the first additional constraint on the stereotype «subQuantityOf».

```
context subQuantityOf
inv subQuantityOfConstraint2b*: self.
    isImmutablePart = true -- Extension of the
    second constraint of the metaclass
    subQuantityOf.
```

- In a relationship stereotyped as «subQuantityOf», the Properties related to the parts must have the meta-attribute isReadOnly = true, because these parts are always immutable (*i.e.*, the meta-attribute isImmutablePart = true).
  - Formalized as an OCL invariant verified in live verification:

    Listing 89: OCL expression for the second additional constraint on the stereotype «subQuantityOf».

    ```
    context Property
    inv subQuantityOfConstraint2c*: if self.target.
        oclIsKindOf(subQuantityOf) then self.isReadOnly
         else true endif -- Extension of the second
        constraint of the metaclass subQuantityOf.
    ```

A.3 Definition of Additional EOperations in OCL

This section documents a set of OCL EOperations created mainly to: (i) support some of the previous OCL invariants; (ii) support the OCL derivations for some derived EAttributes and EReferences of the Ecore OntoUML metamodel; and (iii) automatically calculate the values of a number of meta-attributes, such as the cardinalities of the «material» associations and Derivation relationships, as shown in the Fig. 41.

Additional EOperations

Listing 90: OCL expression for the EOperation `allSuperTypes()`.

```
context Element::allSuperTypes():Bag(Element)
body: if self.oclIsKindOf(Classifier) then (if self.
    oclAsType(Classifier).generalization->forAll(x | x.
    oclIsUndefined()) then Set{} else Set{self.oclAsType(
    Classifier).generalization->collect(x | x.general),
    self.oclAsType(Classifier).generalization->collect(x |
    if x.general.oclIsKindOf(Classifier) then x.general.
```

```
    allSuperTypes() else Set{} endif)}->flatten() endif)
    else Set{} endif
```

Listing 91: OCL expression for the EOperation allSubTypes().

```
context Element::allSubTypes():Bag(Element)
body: let generalizations : Set(Generalization) =
    Generalization.allInstances()->select(x | x.general =
    self) in (if self.oclIsKindOf(Classifier) then (if
    generalizations->forAll(y | y.oclIsUndefined()) then
    Set{} else Set{generalizations->collect(y | y.specific)
    , generalizations->collect(y | if y.specific.
    oclIsKindOf(Classifier) then y.specific.allSubTypes()
    else Set{} endif)}->flatten() endif) else Set{} endif)
```

Listing 92: OCL expression for the EOperation isConected(x:Element).

```
context Element::isConected(x:Element):EBoolean
body: if self.oclIsKindOf(Relationship) then if self.
    oclAsType(Relationship).relatedElement->forAll(y | y.
    oclIsUndefined()) then false else if self.oclAsType(
    Relationship).relatedElement->exists(z | if z.
    oclIsKindOf(Property) then (z.oclAsType(Property).
    endType = x) else false endif) then true else self.
    oclAsType(Relationship).relatedElement->exists(w | if w
    .oclIsKindOf(Property) then w.oclAsType(Property).
    endType.isConected(x) else false endif) endif endif
    else false endif
```

Listing 93: OCL expression for the EOperation subInstanceType(x:Element).

```
context Element::subInstanceType(x:Element):EBoolean
body: self.allSuperTypes()->includes(x)
```

Listing 94: OCL expression for the EOperation subMetaTypeKind().

```
context Element::subMetaTypeKind():EBoolean
body: if self.oclIsKindOf(Kind) then true else self.
    allSuperTypes()->exists(x | x.oclIsKindOf(Kind)) endif
```

Listing 95: OCL expression for the EOperation `subMetaTypeCollective()`.

```
context Element::subMetaTypeCollective():EBoolean
body: if self.oclIsKindOf(Collective) then true else self.
     allSuperTypes()->exists(x | x.oclIsKindOf(Collective))
     endif
```

Listing 96: OCL expression for the EOperation `subMetaTypeQuantity()`.

```
context Element::subMetaTypeQuantity():EBoolean
body: if self.oclIsKindOf(Quantity) then true else self.
     allSuperTypes()->exists(x | x.oclIsKindOf(Quantity))
     endif
```

Listing 97: OCL expression for the EOperation `hasFunctionalComplexesInstances()`.

```
context Element::hasFunctionalComplexesInstances():
     EBoolean
body: if self.oclIsKindOf(SortalClass) then self.
     subMetaTypeKind() else if self.oclIsKindOf(MixinClass)
     then Element.allInstances()->forAll(x | x.
     subInstanceType(self) implies not (x.
     subMetaTypeQuantity() or x.subMetaTypeCollective()))
     else false endif endif
```

Listing 98: OCL expression for the EOperation `hasCollectivesInstances()`.

```
context Element::hasCollectivesInstances():EBoolean
body: if self.oclIsKindOf(SortalClass) then self.
     subMetaTypeCollective() else if self.oclIsKindOf(
     MixinClass) then Element.allInstances()->forAll(x | x.
     subInstanceType(self) implies not (x.subMetaTypeKind()
     or x.subMetaTypeQuantity())) else false endif endif
```

Listing 99: OCL expression for the EOperation `hasQuantitiesInstances()`.

```
context Element::hasQuantitiesInstances():EBoolean
body: if self.oclIsKindOf(SortalClass) then self.
     subMetaTypeQuantity() else if self.oclIsKindOf(
     MixinClass) then Element.allInstances()->forAll(x | x.
     subInstanceType(self) implies not (x.subMetaTypeKind()
     or x.subMetaTypeCollective())) else false endif endif
```

Listing 100: OCL expression for the EOperation deriveUpperMaterialAssociationExt1().

```
context MaterialAssociation::
    deriveUpperMaterialAssociationExt1():EInt
body: let der:Derivation = Derivation.allInstances()->
    select(x | x.source->any(true).oclAsType(Property).
    endType = self)->any(true), matext1:Type = self.
    associationEnd->at(1).oclAsType(Property).endType.
    oclAsType(Type), matext2:Type = self.associationEnd->at
    (2).oclAsType(Property).endType.oclAsType(Type) in (let
     rel:Relator = der.target->any(true).oclAsType(Property
    ).endType.oclAsType(Relator) in (let med1:Set(Mediation
    ) = Mediation.allInstances()->select(x | x.source->
    exists(y | y.oclAsType(Property).endType = rel) and x.
    target->exists(y | y.oclAsType(Property).endType =
    matext1)), med2:Set(Mediation) = Mediation.allInstances
    ()->select(x | x.source->exists(y | y.oclAsType(
    Property).endType = rel) and x.target->exists(y | y.
    oclAsType(Property).endType = matext2)) in (let
    med1targetupper: Integer = med1.target->any(true).
    oclAsType(Property).upper, med2sourceupper: Integer =
    med2.source->any(true).oclAsType(Property).upper in (if
     ((med2sourceupper = -1) or (med1targetupper = -1))
    then (-1) else (med2sourceupper*med1targetupper) endif)
    )))
```

Listing 101: OCL expression for the EOperation deriveUpperMaterialAssociationExt2().

```
context MaterialAssociation::
    deriveUpperMaterialAssociationExt2():EInt
body: let der:Derivation = Derivation.allInstances()->
    select(x | x.source->any(true).oclAsType(Property).
    endType = self)->any(true), matext1:Type = self.
    associationEnd->at(1).oclAsType(Property).endType.
    oclAsType(Type), matext2:Type = self.associationEnd->at
    (2).oclAsType(Property).endType.oclAsType(Type) in (let
     rel:Relator = der.target->any(true).oclAsType(Property
    ).endType.oclAsType(Relator) in (let med1:Set(Mediation
    ) = Mediation.allInstances()->select(x | x.source->
    exists(y | y.oclAsType(Property).endType = rel) and x.
    target->exists(y | y.oclAsType(Property).endType =
    matext1)), med2:Set(Mediation) = Mediation.allInstances
    ()->select(x | x.source->exists(y | y.oclAsType(
    Property).endType = rel) and x.target->exists(y | y.
    oclAsType(Property).endType = matext2)) in (let
    med1sourceupper: Integer = med1.source->any(true).
    oclAsType(Property).upper, med2targetupper: Integer =
    med2.target->any(true).oclAsType(Property).upper in (if
     ((med1sourceupper = -1) or (med2targetupper = -1))
    then (-1) else (med1sourceupper*med2targetupper) endif)
    )))
```

Listing 102: OCL expression for the EOperation `existsDerivationConnected()`.

```
context MaterialAssociation::existsDerivationConnected():
    EBoolean
body: not Derivation.allInstances()->select(x | x.source->
    any(true).oclAsType(Property).endType = self)->isEmpty
    ()
```

Listing 103: OCL expression for the EOperation `deriveLowerDerivation()`.

```
context Derivation::deriveLowerDerivation():EInt
body: let mat:MaterialAssociation = self.source->any(true)
    .oclAsType(Property).endType.oclAsType(
    MaterialAssociation), rel:Relator = self.target->any(
    true).oclAsType(Property).endType.oclAsType(Relator) in
     (let matext1:Type = mat.associationEnd->at(1).
    oclAsType(Property).endType.oclAsType(Type), matext2:
    Type = mat.associationEnd->at(2).oclAsType(Property).
    endType.oclAsType(Type) in (let med1:Set(Mediation) =
    Mediation.allInstances()->select(x | x.source->exists(y
     | y.oclAsType(Property).endType = rel) and x.target->
    exists(y | y.oclAsType(Property).endType = matext1)),
    med2:Set(Mediation) = Mediation.allInstances()->select(
    x | x.source->exists(y | y.oclAsType(Property).endType
     = rel) and x.target->exists(y | y.oclAsType(Property).
     endType = matext2)) in (let med1targetlower: Integer =
     med1.target->any(true).oclAsType(Property).lower,
     med2targetlower: Integer = med2.target->any(true).
     oclAsType(Property).lower in (if ((med1targetlower =
     -1) or (med2targetlower = -1)) then (-1) else (
     med1targetlower*med2targetlower) endif))))
```

Listing 104: OCL expression for the EOperation `deriveUpperDerivation()`.

```
context Derivation::deriveUpperDerivation():EInt
body: let mat:MaterialAssociation = self.source->any(true)
     .oclAsType(Property).endType.oclAsType(
     MaterialAssociation), rel:Relator = self.target->any(
     true).oclAsType(Property).endType.oclAsType(Relator) in
      (let matext1:Type = mat.associationEnd->at(1).
     oclAsType(Property).endType.oclAsType(Type), matext2:
     Type = mat.associationEnd->at(2).oclAsType(Property).
     endType.oclAsType(Type) in (let med1:Set(Mediation) =
     Mediation.allInstances()->select(x | x.source->exists(y
      | y.oclAsType(Property).endType = rel) and x.target->
     exists(y | y.oclAsType(Property).endType = matext1)),
     med2:Set(Mediation) = Mediation.allInstances()->select(
     x | x.source->exists(y | y.oclAsType(Property).endType
     = rel) and x.target->exists(y | y.oclAsType(Property).
     endType = matext2)) in (let med1targetupper: Integer =
     med1.target->any(true).oclAsType(Property).upper,
     med2targetupper: Integer = med2.target->any(true).
     oclAsType(Property).upper in (if ((med1targetupper =
     -1) or (med2targetupper = -1)) then (-1) else (
     med1targetupper*med2targetupper) endif))))
```

A.4 Definition of Derived EReferences in OCL

This section documents a set of OCL derivations that specify how the derived meta-relations of the OntoUML metamodel get their values.

Definition of Derived EAttributes and EReferences

Listing 105: OCL expression for the derived EReference “attribute” of the metaclass Classifier.

```
context Classifier::attribute:Bag(Property)
derive: DatatypeRelationship.allInstances()->select(x | if
    x.source->forAll(y | y.oclIsKindOf(Property)) then x.
    source->exists(y | y.oclAsType(Property).endType = self
    ) else false endif)->collect(x | if x.target->forAll(y
    | y.oclIsKindOf(Property)) then x.target.oclAsType(
    Property) else null endif)
```

Listing 106: OCL expression for the derived EReference “general” of the metaclass Classifier.

```
context Classifier::general:Bag(Classifier)
derive: self.allSuperTypes()
```

Listing 107: OCL expression for the derived EReference “generalization” of the metaclass Classifier.

```
context Classifier::generalization:Bag(Generalization)
derive: Generalization.allInstances()->select(x | x.
    specific = self)
```

Listing 108: OCL expression for the derived EReference “specific” of the metaclass Generalization.

```
context Generalization::specific:Classifier
derive: self.target->any(x | x.oclIsKindOf(Classifier))
```

Listing 109: OCL expression for the derived EReference “general” of the metaclass Generalization.

```
context Generalization::general:Classifier
derive: self.source->any(x | x.oclIsKindOf(Classifier))
```

Listing 110: OCL expression for the derived EReference “endType” of the metaclass Property.

```
context Property::endType:Type
```

```
derive: if self.source->notEmpty() then (if self.source.
     sourceAux1->forAll(x | x.oclIsKindOf(Type)) then self.
     source.sourceAux1.oclAsType(Type)->any(true) else null
     endif) else if self.target->notEmpty() then (if self.
     target.targetAux1->forAll(x | x.oclIsKindOf(Type)) then
      self.target.targetAux1.oclAsType(Type)->any(true) else
      null endif) else if (self.associationEndPositionAux =
     1) then self.associationEnd.associationEndAux1->any(
     true) else if (self.associationEndPositionAux = 2) then
      self.associationEnd.associationEndAux2->any(true) else
      null endif endif endif endif
```

Listing 111: OCL expression for the derived EReference “source” of the metaclass Property.

```
context Property::source:DirectedBinaryRelationship
derive: DirectedBinaryRelationship.allInstances()->any(x |
      x.source->includes(self) or x.sourceAux2->includes(
     self))
```

Listing 112: OCL expression for the derived EReference “target” of the metaclass Property.

```
context Property::target:DirectedBinaryRelationship
derive: DirectedBinaryRelationship.allInstances()->any(x |
      x.target->includes(self) or x.targetAux2->includes(
     self))
```

Listing 113: OCL expression for the derived EReference “relatedElement” of the metaclass Relationship.

```
context Relationship::relatedElement:Bag(Element)
derive: if self.oclIsKindOf(Association) then self.
     oclAsType(Association).associationEnd else if self.
     oclIsKindOf(DirectedRelationship) then Set{self.
     oclAsType(DirectedRelationship).source, self.oclAsType(
     DirectedRelationship).target}->flatten() else null
     endif endif
```

A.5 Definition of the Automatic Calculation of some Meta-Attributes' Values in OCL

This section documents a set of OCL expressions for the automatic initialization or modification of a number of meta-attributes' values, as well as the explanations of the reasons of the pertinence of each of those meta-attributes in the two categories of meta-attributes shown in subsection 4.3.3.

The meta-attributes in the first category are:

- isAbstract (of the metaclass Classifier): The meta-attribute isAbstract must always have the value "true" for instances of the metaclass Mixin Class (due to the second syntactical constraint for the metaclass Mixin Class, shown in page 39 and formalized in OCL in Listing 27). Therefore, we initialize isAbstract with "true" in the creation of a Mixin Class;
- lower (of the metaclass MultiplicityElement):
  - For Mediation relationships, the meta-attribute lower of the Property in the association end connected to the mediated universal (the target association end) must be at least "1" (due to the second syntactical constraint for the metaclass Mediation, shown in page 48 and formalized in OCL in Listing 36). Therefore, we initialize lower with "1" in the creation of a Property which target meta-relation points to a Mediation relationship;
  - For Mediation relationships, the meta-attribute lower of the Property in the association end connected to the relator universal (the source association end) must be at least "1" (due to the fourth syntactical constraint for the metaclass Mediation, shown in page 48 and formalized in OCL in Listing 38). Therefore, we initialize lower with "1" in the creation of a Property which source meta-relation points to a Mediation relationship;
  - For Characterization relationships, the meta-attribute lower of the Property in the association end connected to the characterized universal (the target association end) must be "1" (due to the second syntactical constraint for the metaclass Characterization, shown in page 48 and formalized in OCL in Listing 41). Therefore, we initialize lower with "1" in the creation of a Property which target meta-relation points to a Characterization relationship;
  - For Characterization relationships, the meta-attribute lower of the Property in the association end connected to the characterizing quality universal (the source

association end) must be at least “1” (due to the third syntactical constraint for the metaclass Characterization, shown in page 48 and formalized in OCL in Listing 42). Therefore, we initialize lower with “1” in the creation of a Property which source meta-relation points to a Characterization relationship;

  - For Derivation relationships, the meta-attribute lower of the Property in the black circle end of the Derivation relation (the target association end) must be “1” (due to the third syntactical constraint for the metaclass Derivation, shown in page 49 and formalized in OCL in Listing 47). Therefore, we initialize lower with “1” in the creation of a Property which target meta-relation points to a Derivation relationship;
  - For Derivation relationships, the meta-attribute lower of the Property in the association end connected to the Material Association (the source association end) is a product of the cardinality constraints of the Mediation relations of the Relator Universal that this Material Association derives from (as shown in Fig. 41). However, since Mediation relationships require a minimum cardinality of one on both of its association ends, then the minimum cardinality on the Material Association end of a Derivation relation must also be ≥ 1 (due to the fifth syntactical constraint for the metaclass Derivation, shown in page 49 and formalized in OCL in Listing 49). Therefore, we initialize lower with “1” in the creation of a Property which source meta-relation points to a Derivation relationship;
  - For Material Associations, the meta-attribute lower of the Properties in its association ends are derived from the cardinality constraints of the Mediation relations of the Relator Universal that this Material Association is derived from (as shown in Fig. 41). However, since Mediation relationships require a minimum cardinality of one on both of its association ends, then the minimum cardinality constraint on each end of the derived Material Association must also be ≥ 1 (due to the second syntactical constraint for the metaclass Material Association, shown in page 50 and formalized in OCL in Listing 51). Therefore, we initialize lower with “1” in the creation of a Property which associationEnd meta-relation points to a Material Association;
  - For Datatype Relationships, the meta-attribute lower of the Property in the target association end must be “1” (due to the first syntactical constraint for the metaclass Property, shown in page 50 and formalized in OCL in Listing 53). Therefore, we initialize lower with “1” in the creation of a Property which target meta-relation points to a Datatype Relationship;

- upper (of the metaclass MultiplicityElement):

  - For Characterization relationships, the meta-attribute upper of the Property in the association end connected to the characterized universal (the target association end) must be "1" (due to the second syntactical constraint for the metaclass Characterization, shown in page 48 and formalized in OCL in Listing 41). Therefore, we initialize lower with "1" in the creation of a Property which target meta-relation points to a Characterization relationship;
  - For Derivation relationships, the meta-attribute upper of the Property in the black circle end of the Derivation relation (the target association end) must be "1" (due to the third syntactical constraint for the metaclass Derivation, shown in page 49 and formalized in OCL in Listing 47). Therefore, we initialize upper with "1" in the creation of a Property which target meta-relation points to a Derivation relationship;
  - For subQuantityOf relationships, the meta-attribute upper of the Property in the association end connected to the part (the target association end) must be "1" (due to the third syntactical constraint for the metaclass subQuantityOf, shown in page 56 and formalized in OCL in Listing 59). Therefore, we initialize upper with "1" in the creation of a Property which target meta-relation points to a subQuantityOf relationship;
  - For subCollectionOf relationships, the meta-attribute upper of the Property in the association end connected to the part (the target association end) must be "1" (due to the second syntactical constraint for the metaclass subCollectionOf, shown in page 57 and formalized in OCL in Listing 62). Therefore, we initialize upper with "1" in the creation of a Property which target meta-relation points to a subCollectionOf relationship;

- isReadOnly (of the metaclass StructuralFeature):
  - For Mediation relationships, the Property in the target association end must have the meta-attribute isReadOnly = "true" (due to the third syntactical constraint for the metaclass Mediation, shown in page 48 and formalized in OCL in Listing 37). Therefore, we initialize isReadOnly with "true" in the creation of a Property which target meta-relation points to a Mediation relationship;
  - For Characterization relationships, the Property in the association end connected to the characterized universal (the target association end) must have the meta-attribute isReadOnly = "true" (due to the fourth syntactical constraint for the metaclass Characterization, shown in page 48 and formalized in OCL in Listing 43). Therefore,

  we initialize isReadOnly with "true" in the creation of a Property which target meta-relation points to a Characterization relationship;

  - For Derivation relationships, the Property in the black circle end of the Derivation relation (the target association end) must have the meta-attribute isReadOnly = "true" (due to the fourth syntactical constraint for the metaclass Derivation, shown in page 49 and formalized in OCL in Listing 48). Therefore, we initialize isReadOnly with "true" in the creation of a Property which target meta-relation points to a Derivation relationship;

  - For subQuantityOf relationships, the Property in the association end connected to the part (the target association end) must have the meta-attribute isReadOnly = "true" (due to the second additional syntactical constraint for the metaclass subQuantityOf, shown in page 72 and formalized in OCL in Listing 89). Therefore, we initialize isReadOnly with "true" in the creation of a Property which target meta-relation points to a subQuantityOf relationship;

- isDerived (of the metaclass Association): The meta-attribute isDerived must always be true for instances of the metaclass Material Association (due to the third syntactical constraint for the metaclass Material Association, shown in page 50 and formalized in OCL in Listing 52). Therefore, we initialize isDerived with "true" in the creation of a Material Association;

- isShareable (of the metaclass Meronymic): The meta-attribute isShareable must always be "false" for instances of the metaclass subQuantityOf (due to the first syntactical constraint for the metaclass subQuantityOf, shown in page 56 and formalized in OCL in Listing 57). Therefore, we initialize isShareable with "false" in the creation of a subQuantityOf relationship;

- isEssential (of the metaclass Meronymic): The meta-attribute isEssential must always be "true" for instances of the metaclass subQuantityOf because all Quantities are extensional individuals (due to the second syntactical constraint for the metaclass subQuantityOf, shown in page 56 and formalized in OCL in Listing 58). Therefore, we initialize isEssential with "true" in the creation of a subQuantityOf relationship;

- isDerived (of the metaclass Property): For Material Associations, the meta-attribute isDerived of the Properties in its association ends must always be "true" (due to the first additional syntactical constraint for the metaclass Material Association, shown in page 71 and formalized in OCL in Listing 77). Therefore, we initialize isDerived with "true" in the creation of a Property which associationEnd meta-relation points to a Material Association;

- isImmutablePart (of the metaclass Meronymic): The meta-attribute isImmutablePart must always be “true” for instances of the metaclass subQuantityOf because isEssential is always “true” for this individuals (due to the first additional syntactical constraint for the metaclass subQuantityOf, shown in page 72 and formalized in OCL in Listing 88). Therefore, we initialize isImmutablePart with “true” in the creation of a subQuantityOf relationship;

The meta-attributes in the second category are:

- lower (of the metaclass MultiplicityElement): For a Derivation relationship *d*, the meta-attribute lower of the Property in its source association end is systematically calculated from the cardinalities of the Mediation relationships that are relating (i) the Relator in the black circle end (the target association end) of *d* and (ii) a Classifier in an association end of the Material Association that is in the source association end of *d*, as shown in Fig. 41 and implemented in OCL as shown in Listing 103. Therefore, we initialize the value of lower with the value calculated from the *formulæ* shown in Fig. 41, in the creation of a Derivation relationship. Moreover, when the user updates the values of the cardinalities of the related Mediation relationships, we automatically modify the value of lower. Additionally, the first additional syntactical constraint for the metaclass Derivation, shown in page 71 and formalized in OCL in Listing 79, is used to guarantee the consistence of the model when the user manually sets the value of the meta-attribute lower of the Property in the source association end of a Derivation relationship;

- upper (of the metaclass MultiplicityElement):

  - For a Material Association *m*, the meta-attribute upper of the Properties in its association ends are systematically calculated from the cardinalities of the Mediation relationships that are relating (i) the Relator in the black circle end (the target association end) of the Derivation relation connected to *m* and (ii) a Classifier in an association end of *m*, as shown in Fig. 41 and implemented in OCL as shown in Listings 100 and 101. Therefore, when the user creates a Derivation relationship between *m* and a Relator, or when he/she updates the values of the cardinalities of the related Mediation relationships, we automatically modify the value of upper with the value calculated from the *formulæ* shown in Fig. 41. Additionally, the second

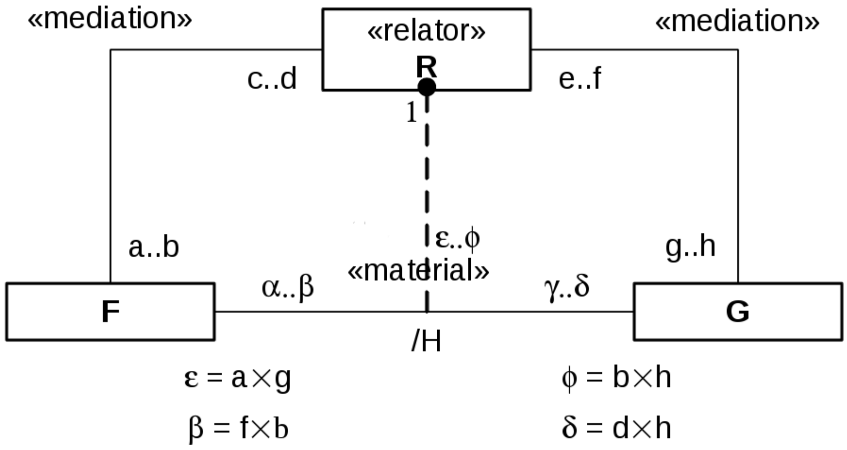


Figure 41: Calculus of the cardinalities (69, p. 533). In this figure, the × operator is the multiplication operator between natural numbers.

additional syntactical constraint for the metaclass Material Association, shown in page 71 and formalized in OCL in Listing 78, is used to guarantee the consistence of the model when the user manually sets the value of the meta-attribute upper;

- For a Derivation relationship *d*, the meta-attribute upper of the Property in its source association end is systematically calculated from the cardinalities of the Mediation relationships that are relating (i) the Relator in the black circle end (the target association end) of *d* and (ii) a Classifier in an association end of the Material Association that is in the source association end of *d*, as shown in Fig. 41 and implemented in OCL as shown in Listing 104. Therefore, we initialize the value of upper with the value calculated from the *formulæ* shown in Fig. 41, in the creation of a Derivation relationship. Moreover, when the user updates the values of the cardinalities of the related Mediation relationships, we automatically modify the value of upper. Additionally, the first additional syntactical constraint for the metaclass Derivation, shown in page 71 and formalized in OCL in Listing 79, is used to guarantee the consistence of the model when the user manually sets the value of the meta-attribute upper of the Property in the source association end of a Derivation relationship;

- isEssential (of the metaclass Meronymic): For every Collective *c*, when the value of the meta-attribute isExtensional is "true", the value of the meta-attribute isEssential of all Meronymic relationships having *c* in their source association end must be "true" (due to the first syntactical constraint for the metaclass Collective, shown in page 37 and formalized in OCL in Listing 21). Therefore, when the user updates the value of isExtensional to "true", we automatically modify the value of isEssential to "true" in all related Meronymic relationships[3]; or if a Meronymic relationship *m* is created having an extensional Collective (isExtensional's value is "true") in its source, the value of the meta-attribute isEssential of *m* is automatically set to "true", as shown in Listing 114;

Listing 114: OCL expression for the initialization of the value of the meta-attribute isEssential of memberOf and subCollectionOf relationships.

```
if self.source->forAll(x | if x.oclIsKindOf(Property
    ) then (if x.oclAsType(Property).endType.
    oclIsKindOf(Collective) then x.oclAsType(Property
    ).endType.oclAsType(Collective).isExtensional
    else false endif) else false endif) then true
    else false endif
```

- isImmutablePart (of the metaclass Meronymic): For every Meronymic relationship, if the value of the meta-attribute isEssential is "true", the value of the meta-attribute isImmutablePart must also be "true" (due to the second syntactical constraint for the metaclass Meronymic, shown in page 55 and formalized in OCL in Listing 55). Therefore, when the user updates the value of isEssential to "true", we automatically modify the value of isImmutablePart to "true";

- isExtensional (of the metaclass Collective): For every memberOf relationship, if the value of the meta-attribute isEssential is "true", the value of the meta-attribute isExtensional of the whole (the Classifier in the source association end) must also be "true" (due to the first syntactical constraint for the metaclass memberOf, shown in page 58 and formalized in OCL in Listing 63). Therefore, when the user updates the value of isEssential to "true", we automatically modify the value of isExtensional to "true" in the whole;

- isReadOnly (of the metaclass StructuralFeature):
  - For every Datatype Relationship, if there is a StructuredDatatype in its source association end, the value of the meta-attribute isReadOnly of the Property in the target association end must be "true" (due to the third additional syntactical constraint for the metaclass Datatype Relationship, shown in page 70 and formalized in OCL in Listing 68). Therefore, when the user creates a Datatype Relationship having a

[3] Only for the memberOf and the subCollectionOf relationships, because a Collective cannot be in an association end of a componentOf or a subQuantityOf relationship due to the first syntactical constraint for the metaclass componentOf, shown in page 56 and formalized in OCL in Listing 56 and the fourth syntactical constraint for the metaclass subQuantityOf, shown in page 57 and formalized in OCL in Listing 60.

StructuredDatatype in its source association end, we automatically modify the value of isReadOnly to “true” in the Property in the target association end, as shown in Listing 115;

Listing 115: OCL expression for the initialization of the value of the meta-attribute isReadOnly of the Property in the target association end of a Datatype Relationship in the creation of the latter.

```
if (self.target.oclIsKindOf(
    DatatypeRelationship)) then (if (self.
    target.source->forAll(x | if (x.oclIsKindOf
    (Property)) then (x.oclAsType(Property).
    endType.oclIsKindOf(StructuralDatatype))
    else false endif)) then true else false
    endif) else false endif
```

- For every Meronymic relationship, if the value of the meta-attribute isImmutablePart is “true”, the value of the meta-attribute isReadOnly in the Property in the target association end must also be “true” (due to the first additional syntactical constraint for the metaclass Meronymic, shown in page 72 and formalized in OCL in Listing 81). Therefore, when the user updates the value of isImmutablePart to “true”, we automatically modify the value of isReadOnly to “true” in the Property in the target association end;
- For every Meronymic relationship, if the value of the meta-attribute isImmutableWhole is “true”, the value of the meta-attribute isReadOnly in the Property in the source association end must also be “true” (due to the third additional syntactical constraint for the metaclass Meronymic, shown in page 72 and formalized in OCL in Listing 83). Therefore, when the user updates the value of isImmutableWhole to “true”, we automatically modify the value of isReadOnly to “true” in the Property in the source association end;

- isImmutableWhole (of the metaclass Meronymic): For every Meronymic relationship, if the value of the meta-attribute isInseparable is “true”, the value of the meta-attribute isImmutableWhole must also be “true” (due to the second additional syntactical constraint for the metaclass Meronymic, shown in page 72 and formalized in OCL in Listing 82). Therefore, when the user updates the value of isInseparable to “true”, we automatically modify the value of isImmutableWhole to “true”;

APPENDIX B – Implementing The Mapping as an ATL Model Transformation

The purpose of this chapter is to document our implementation of all the mappings from OntoUML to Alloy, which were discussed in section 5.5 as an ATL (see section 2.2.4) automatic transformation. This transformation receives the OntoUML metamodel (Fig. 19) and a source OntoUML model as inputs and creates an Alloy specification as its output.

As we had no metamodel of the Alloy language, here, we make use of the imperative constructs of ATL in order to transform an OntoUML model into an Alloy specification. Therefore, we do not map the OntoUML metamodel to an Alloy metamodel, instead, we use the ATL engine to search for certain patterns in OntoUML models and imperatively build the Alloy counterparts.

This ATL transformation is show in Listing 116 and is a FOSS licensed under GPLv3 (see annex A).

For explanations on how to install and use this transformation within the ATL Eclipse plug-in, see appendix D.

Listing 116: The ATL transformation.

```
-- @path OntoUML=/OntoUML2Alloy/OntoUML.ecore

module OntoUML2Alloy; -- Module Template
create OUT : OntoUML refining IN : OntoUML;

helper def: path : String = '/OntoUML2Alloy/specification.
    als';
helper def: signature_text : String = '';
helper def: relation_relations : String = '';
helper def: directedbinaryrelationship_generalization() :
    String = OntoUML!DirectedBinaryRelationship.
    allInstances()->select(x | not x.source->exists(y | if
    y.oclIsKindOf(OntoUML!Property) then (thisModule.
    derive_endType(y).oclIsKindOf(OntoUML!
    StructuralDatatype)) else false endif))->iterate(x;
    directedbinaryrelationship: String = '' | let gen :
    OrderedSet(OntoUML!Classifier) = thisModule.
    derive_generalization(x)->collect(y | thisModule.
    derive_general(y)) in (if (gen.size() > 0) then (
    directedbinaryrelationship + '  ' + x.name + ' in ' +
    thisModule.names_disjunction_set2(gen) + '\n') else
    directedbinaryrelationship endif));
helper def: association_generalization() : String =
    OntoUML!Association.allInstances()->iterate(x;
    association: String = '' | let gen : OrderedSet(OntoUML
    !Classifier) = thisModule.derive_generalization(x)->
    collect(y | thisModule.derive_general(y)) in (if (gen.
    size() > 0) then (association + ' ' + x.name + ' in ' +
     thisModule.names_disjunction_set2(gen) + '\n') else
    association endif));
helper def: relation_generalization() : String =
    thisModule.directedbinaryrelationship_generalization()
    + thisModule.association_generalization();
helper def: signature_names() : String = thisModule.
    names_union_set(OntoUML!SubstanceSortal.allInstances()
    ->union(OntoUML!MomentClass.allInstances()));
helper def: relation_phases() : String = OntoUML!
    GeneralizationSet.allInstances()->select(x | if (x.
    generalization.size() > 0) then (if (x.generalization->
    forAll(y | thisModule.derive_specific(y).oclIsKindOf(
    OntoUML!Phase))) then true else false endif) else false
     endif)->iterate(x; phase_generalizationset: String = '
    ' | phase_generalizationset + ' disj ' + thisModule.
    names_list(x.generalization->iterate(y; phases:
    OrderedSet(OntoUML!Phase) = OrderedSet{} | phases.
    including(thisModule.derive_specific(y)))) + ': set ' +
     thisModule.derive_general(x.generalization->any(y |
    true)).name + (if (thisModule.derive_general(x.
    generalization->any(y | true)).oclIsKindOf(OntoUML!
    SubstanceSortal)) then ' :> domain_of_quantification,\n
    ' else ',\n' endif));
helper def: relation_roles() : String = OntoUML!Role.
    allInstances()->iterate(x; role: String = '' | let gen
    : OrderedSet(OntoUML!Generalization) = thisModule.
    derive_generalization(x)->collect(y | thisModule.
    derive_general(y)) in (role + '   ' + x.name + ': set '
    + thisModule.names_disjunction_set2(gen) + ',\n'));
helper def: relation_mixins() : String = OntoUML!Mixin.
    allInstances()->iterate(x; mixin: String = '' | mixin +
     '   ' + x.name + ': set ' + thisModule.
    subTypes_name_union_set_special_rigid_sortal_subtypes(x
    ) + ',\n');
helper def: relation_rolemixins() : String = OntoUML!
    RoleMixin.allInstances()->iterate(x; rolemixin: String
    = '' | rolemixin + '   ' + x.name + ': set ' +
    thisModule.subTypes_name_union_set(x) + ',\n');
helper def: signature_declarations_world() : String = '
    abstract sig World {\n domain_of_quantification: some (
    ' + thisModule.signature_names() + '),\n' + thisModule.
    relation_phases() + thisModule.relation_roles() +
    thisModule.relation_mixins() + thisModule.
    relation_rolemixins() + thisModule.relation_relations +
     '}';
helper def: signature_facts_world : String = '\n' +
    thisModule.relation_generalization();
helper def: signature_world() : String = thisModule.
    signature_declarations_world() + if (thisModule.
    signature_facts_world <> '') then ('{' + thisModule.
    signature_facts_world + '}') else '' endif;
helper def: function_text : String = '';
helper def: fact_text : String = '';
helper def: pairwise_disjoint_Classifiers : String = '';
helper def: command_text : String = '';
helper def: final_text() : String = thisModule.
      signature_text + '\n' + thisModule.function_text + '\n'
       + thisModule.signature_world() + '\n' + thisModule.
      pairwise_disjoint_Classifiers + '\n' + thisModule.
      fact_text + '\n' + thisModule.command_text;

helper def: derive_source(p: OntoUML!Property) : OntoUML!
      DirectedBinaryRelationship = OntoUML!
      DirectedBinaryRelationship.allInstances()->any(x | x.
      source->includes(p) or x.sourceAux2->includes(p));
helper def: derive_target(p: OntoUML!Property) : OntoUML!
      DirectedBinaryRelationship = OntoUML!
      DirectedBinaryRelationship.allInstances()->any(x | x.
      target->includes(p) or x.targetAux2->includes(p));
helper def: derive_endType(p: OntoUML!Property) : OntoUML!
      Type = if (not thisModule.derive_source(p)->
      oclIsUndefined()) then (if thisModule.derive_source(p).
      sourceAux1->forAll(x | x.oclIsKindOf(OntoUML!Type))
      then thisModule.derive_source(p).sourceAux1->any(x |
      true) else OclUndefined endif) else if (not thisModule.
      derive_target(p)->oclIsUndefined()) then (if thisModule
      .derive_target(p).targetAux1->forAll(x | x.oclIsKindOf(
      OntoUML!Type)) then thisModule.derive_target(p).
      targetAux1->any(x | true) else OclUndefined endif) else
       if (p.associationEndPositionAux = 1) then p.
      associationEnd.associationEndAux1->any(x | true) else
      if (p.associationEndPositionAux = 2) then p.
      associationEnd.associationEndAux2->any(x | true) else
      OclUndefined endif endif endif endif;
helper def: derive_specific(g: OntoUML!Generalization) :
      OntoUML!Classifier = g.target->any(x | x.oclIsKindOf(
      OntoUML!Classifier));
helper def: derive_general(g: OntoUML!Generalization) :
    OntoUML!Classifier = g.source->any(x | x.oclIsKindOf(
    OntoUML!Classifier));
helper def: derive_generalization(c: OntoUML!Classifier) :
     OrderedSet(OntoUML!Generalization) = OntoUML!
    Generalization.allInstances()->select(x | thisModule.
    derive_specific(x) = c);
helper def: allSuperTypes(c: OntoUML!Classifier) :
    OrderedSet(OntoUML!Element) = if thisModule.
    derive_generalization(c)->forAll(x | x.oclIsUndefined()
    ) then Set{} else Set{thisModule.derive_generalization(
    c)->collect(x | thisModule.derive_general(x)),
    thisModule.derive_generalization(c)->collect(x | if
    thisModule.derive_general(x).oclIsKindOf(OntoUML!
    Classifier) then thisModule.allSuperTypes(thisModule.
    derive_general(x)) else Set{} endif)}->flatten() endif;

helper def: names_list(os: OrderedSet(OntoUML!NamedElement
    )) : String = let names: String = os->iterate(x; names:
     String = '' | names + x.name + ', ') in (if (names.
    size() > 0) then names.substring(1,(names.size()-2))
    else '' endif);
helper def: names_union_set(os: OrderedSet(OntoUML!
    NamedElement)) : String = let names: String = os->
    iterate(x; names: String = '' | names + x.name + ' + ')
     in (if (names.size() > 0) then names.substring(1,(
    names.size()-3)) else '' endif);
helper def: names_union_set_special_rigid_sortal_subtypes(
    os: OrderedSet(OntoUML!NamedElement)) : String = let
    names: String = os->iterate(x; names: String = '' | if
    (x.oclIsKindOf(OntoUML!RigidSortalClass)) then (names +
     x.name + ':>domain_of_quantification + ') else (names
    + x.name + ' + ') endif) in (if (names.size() > 0) then
     names.substring(1,(names.size()-3)) else '' endif);
helper def: names_disjunction_set(os: OrderedSet(OntoUML!
    NamedElement)) : String = let names: String = os->
    iterate(x; names: String = '' | names + x.name + ' & ') in (if (names.size() > 0) then names.substring(1,(names.size()-3)) else '' endif);
helper def: domain_function(e: OntoUML!Element): String = if (e.oclIsKindOf(OntoUML!SubstanceSortal) or e.oclIsKindOf(OntoUML!Relator) or e.oclIsKindOf(OntoUML!Mode)) then ':>domain_of_quantification' else '' endif;
helper def: names_disjunction_set2(os: OrderedSet(OntoUML!NamedElement)) : String = let names: String = os->iterate(x; names: String = '' | names + x.name + thisModule.domain_function(x) + ' & ') in (if (names.size() > 0) then names.substring(1,(names.size()-3)) else '' endif);

helper def: superTypes_set(e: OntoUML!Element) : OrderedSet(OntoUML!Element) = OntoUML!Generalization.allInstances()->select(x | x.target->any(y | true) = e)->collect(x | x.source)->flatten();
helper def: superTypes_name_list(e: OntoUML!Element) : String = thisModule.names_list(thisModule.superTypes_set(e));
helper def: superTypes_name_union_set(e: OntoUML!Element) : String = thisModule.names_union_set(thisModule.superTypes_set(e));
helper def: superTypes_name_disjunction_set(e: OntoUML!Element) : String = thisModule.names_disjunction_set(thisModule.superTypes_set(e));
helper def: superTypes_set_equality(e1: OntoUML!Element, e2: OntoUML!Element) : Boolean = thisModule.superTypes_set(e1).asSet() = thisModule.superTypes_set(e2).asSet();
helper def: classifiers_with_same_supertypes : Set(Set(OntoUML!Element)) = let c: Set(OntoUML!Classifier) = OntoUML!Classifier.allInstances() in (c->iterate(x; acc : Set(Set(OntoUML!Element)) = Set{} | acc->including(c
      ->select(y | thisModule.superTypes_set_equality(x,y))->
      asSet())));
helper def: subTypes_set(e: OntoUML!Element) : OrderedSet(
      OntoUML!Element) = OntoUML!Generalization.allInstances
      ()->select(x | x.source->any(y | true) = e)->collect(x
      | x.target)->flatten();
helper def: subTypes_name_union_set(e: OntoUML!Element) :
      String = thisModule.names_union_set(thisModule.
      subTypes_set(e));
helper def:
      subTypes_name_union_set_special_rigid_sortal_subtypes(e
      : OntoUML!Element) : String = thisModule.
      names_union_set_special_rigid_sortal_subtypes(
      thisModule.subTypes_set(e));
helper def: generalization_Sets_of_superTypes_of(c:
      OntoUML!Classifier) : OrderedSet(OntoUML!
      GeneralizationSet) = OntoUML!Generalization.
      allInstances()->select(x | x.target->any(y | true) = c)
      ->collect(x | x.generalizationSet)->flatten();
helper def: generalization_Sets_of_subKinds_subTypes_of(c:
       OntoUML!Classifier) : OrderedSet(OntoUML!
      GeneralizationSet) = OntoUML!Generalization.
      allInstances()->select(x | (x.source->any(y | true) = c
      ) and (x.target->forAll(y | y.oclIsKindOf(OntoUML!
      SubKind))))->collect(x | x.generalizationSet)->flatten
      ();
helper def: generalization_relational_constraint(set:
      OrderedSet(OntoUML!Classifier)) : String = set->iterate
      (x; str: String = '' | let gen : OrderedSet(OntoUML!
      Generalization) = thisModule.derive_generalization(x)->
      collect(y | thisModule.derive_general(y)) in (if (gen.
      size() > 0) then (str + ' ' + x.name + ' in ' +
      thisModule.names_disjunction_set2(gen) + '\n') else str
       endif)); -- For a set of relations.
helper def: top_level_rigid_sortals_connected_on_source(d:
      OntoUML!DirectedRelationship) : OrderedSet(OntoUML!
     Classifier) = thisModule.allSuperTypes(thisModule.
     derive_endType(d.source->any(x | true)))->including(
     thisModule.derive_endType(d.source->any(x | true)))->
     select(x | x.oclIsKindOf(OntoUML!SubstanceSortal) or x.
     oclIsKindOf(OntoUML!MomentClass) or x.oclIsKindOf(
     OntoUML!Datatype));
helper def: top_level_rigid_sortals_connected_on_target(d:
      OntoUML!DirectedRelationship) : OrderedSet(OntoUML!
     Classifier) = thisModule.allSuperTypes(thisModule.
     derive_endType(d.target->any(x | true)))->including(
     thisModule.derive_endType(d.target->any(x | true)))->
     select(x | x.oclIsKindOf(OntoUML!SubstanceSortal) or x.
     oclIsKindOf(OntoUML!MomentClass) or x.oclIsKindOf(
     OntoUML!Datatype));
helper def:
     top_level_rigid_sortals_connected_on_associationEnd(d:
     OntoUML!DirectedRelationship, i : Integer) : OrderedSet
     (OntoUML!Classifier) = thisModule.allSuperTypes(
     thisModule.derive_endType(d.associationEnd->at(i)))->
     including(thisModule.derive_endType(d.associationEnd->
     at(i)))->select(x | x.oclIsKindOf(OntoUML!
     SubstanceSortal) or x.oclIsKindOf(OntoUML!MomentClass)
     or x.oclIsKindOf(OntoUML!Datatype));

helper def: is_top_level(e: OntoUML!Element) : Boolean =
     if (e.oclIsKindOf(OntoUML!SubstanceSortal) or e.
     oclIsKindOf(OntoUML!MomentClass)) then true else false
     endif;
helper def: name_side_source(e: OntoUML!Element) : String
     = let side: OntoUML!Element = thisModule.derive_endType
     (e.source->any(x | true)) in (if (thisModule.
     is_top_level(side)) then (side.name + ':>
     domain_of_quantification') else (side.name) endif);
helper def: name_side_target(e: OntoUML!Element) : String
```

```
      = let side: OntoUML!Element = thisModule.derive_endType
      (e.target->any(x | true)) in (if (thisModule.
      is_top_level(side)) then (side.name + ':>
      domain_of_quantification') else (side.name) endif);
helper def: name_side_associationEnd(e: OntoUML!Element, i
      : Integer) : String = let side: OntoUML!Element =
      thisModule.derive_endType(e.associationEnd->at(i)) in (
      if (thisModule.is_top_level(side)) then (side.name + '
      :>domain_of_quantification') else (side.name) endif);

helper def: cardinality(p: OntoUML!Property) : String = if
       ((p.lower = 0) and (p.upper = 1)) then 'lone' else (if
       ((p.lower = 1) and (p.upper = 1)) then 'one' else (if
      ((p.lower = 1) and (p.upper = 0-1)) then 'some' else '
      set' endif) endif) endif;
helper def: cardinality_fact1(name_relation: String,
      name_side: String, p: OntoUML!Property) : String = if (
      p.lower = 0) then (if (p.upper = 0-1) then '' else ('
      all x: ' + name_side + ' | #' + name_relation + '.x <=
      ' + p.upper.toString()) endif) else (if (p.upper = 0-1)
       then (' all x: ' + name_side + ' | #' + name_relation
       + '.x >= ' + p.lower.toString()) else (if (p.lower <>
      p.upper) then (' all x: ' + name_side + ' | (#' +
      name_relation + '.x >= ' + p.lower.toString() + ') and
      (#' + name_relation + '.x <= ' + p.upper.toString() + '
      )') else (' all x: ' + name_side + ' | #' +
      name_relation + '.x = ' + p.lower.toString()) endif)
      endif) endif;
helper def: cardinality_fact2(name_relation: String,
      name_side: String, p: OntoUML!Property) : String = if (
      p.lower = 0) then (if (p.upper = 0-1) then '' else ('
      all x: ' + name_side + ' | #x.' + name_relation + ' <=
      ' + p.upper.toString()) endif) else (if (p.upper = 0-1)
       then (' all x: ' + name_side + ' | #x.' +
      name_relation + ' >= ' + p.lower.toString()) else (if (
     p.lower <> p.upper) then (’ all x: ’ + name_side + ’ |
      (#x.’ + name_relation + ’ >= ’ + p.lower.toString() +
     ’) and (#x.’ + name_relation + ’ <= ’ + p.upper.
     toString() + ’)’) else (’ all x: ’ + name_side + ’ | #
     x.’ + name_relation + ’ = ’ + p.lower.toString()) endif
     ) endif) endif;
helper def: cardinality_mediation1(name_relation: String,
     p: OntoUML!Property) : String = if (p.lower = 0) then (
     if (p.upper = 0-1) then ’’ else (’ #’ + name_relation +
      ’ <= ’ + p.upper.toString()) endif) else (if (p.upper
     = 0-1) then (’ #’ + name_relation + ’ >= ’ + p.lower.
     toString()) else (if (p.lower <> p.upper) then (’ (#’ +
      name_relation + ’ >= ’ + p.lower.toString() + ’) and
     (#’ + name_relation + ’ <= ’ + p.upper.toString() + ’)’
     ) else (’ #’ + name_relation + ’ = ’ + p.lower.toString
     ()) endif) endif) endif;
helper def: cardinality_mediation2(name_relation: String,
     name_side: String, p: OntoUML!Property) : String = if (
     p.lower = 0) then (if (p.upper = 0-1) then ’’ else (’
     all x: ’ + name_side + ’ | #((’ + name_relation + ’.x)
     :>domain_of_quantification) <= ’ + p.upper.toString())
     endif) else (if (p.upper = 0-1) then (’ all x: ’ +
     name_side + ’ | #((’ + name_relation + ’.x):>
     domain_of_quantification) >= ’ + p.lower.toString())
     else (if (p.lower <> p.upper) then (’ all x: ’ +
     name_side + ’ | (#((’ + name_relation + ’.x):>
     domain_of_quantification) >= ’ + p.lower.toString() + ’
     ) and (#((’ + name_relation + ’.x):>
     domain_of_quantification) <= ’ + p.upper.toString() + ’
     )’) else (’ all x: ’ + name_side + ’ | #((’ +
     name_relation + ’.x):>domain_of_quantification) = ’ + p
     .lower.toString()) endif) endif) endif;
helper def: cardinality_derivation1(name_relation: String,
      name_side: String, p: OntoUML!Property) : String = if
     (p.lower = 0) then (if (p.upper = 0-1) then ’’ else (’
    all x: ' + name_side + ' | #x.' + name_relation + ' <=
    ' + p.upper.toString()) endif) else (if (p.upper = 0-1)
     then ('  all x: ' + name_side + ' | #x.' +
    name_relation + ' >= ' + p.lower.toString()) else (if (
    p.lower <> p.upper) then ('  all x: ' + name_side + ' |
     (#x.' + name_relation + ' >= ' + p.lower.toString() +
    ') and (#x.' + name_relation + ' <= ' + p.upper.
    toString() + ')') else ('  all x: ' + name_side + ' | #
    x.' + name_relation + ' = ' + p.lower.toString()) endif
    ) endif) endif;
helper def: cardinality_derivation2(name_relation: String,
     name_side: String, p: OntoUML!Property) : String = if
    (p.lower = 0) then (if (p.upper = 0-1) then '' else ('
    all x: ' + name_side + ' | #' + name_relation + '.x <=
    ' + p.upper.toString()) endif) else (if (p.upper = 0-1)
     then ('  all x: ' + name_side + ' | #' + name_relation
     + '.x >= ' + p.lower.toString()) else (if (p.lower <>
    p.upper) then ('  all x: ' + name_side + ' | (#' +
    name_relation + '.x >= ' + p.lower.toString() + ') and
    (#' + name_relation + '.x <= ' + p.upper.toString() + '
    )') else ('  all x: ' + name_side + ' | #' +
    name_relation + '.x = ' + p.lower.toString()) endif)
    endif) endif;
helper def: cardinality_characterization1(name_relation:
    String, p: OntoUML!Property) : String = if (p.lower =
    0) then (if (p.upper = 0-1) then '' else ('  #' +
    name_relation + ' <= ' + p.upper.toString()) endif)
    else (if (p.upper = 0-1) then (' #' + name_relation + '
     >= ' + p.lower.toString()) else (if (p.lower <> p.
    upper) then (' (#' + name_relation + ' >= ' + p.lower.
    toString() + ') and (#' + name_relation + ' <= ' + p.
    upper.toString() + ')') else (' #' + name_relation + '
    = ' + p.lower.toString()) endif) endif) endif;
helper def: cardinality_characterization2(name_relation:
    String, name_side: String, p: OntoUML!Property) :
```

```
    String = if (p.lower = 0) then (if (p.upper = 0-1) then
     '' else (' all x: ' + name_side + ' | #((' +
    name_relation + '.x):>domain_of_quantification) <= ' +
    p.upper.toString()) endif) else (if (p.upper = 0-1)
    then ('  all x: ' + name_side + ' | #((' +
    name_relation + '.x):>domain_of_quantification) >= ' +
    p.lower.toString()) else (if (p.lower <> p.upper) then
    ('  all x: ' + name_side + ' | (#((' + name_relation +
    '.x):>domain_of_quantification) >= ' + p.lower.toString
    () + ') and (#((' + name_relation + '.x):>
    domain_of_quantification) <= ' + p.upper.toString() + '
    )') else ('  all x: ' + name_side + ' | #((' +
    name_relation + '.x):>domain_of_quantification) = ' + p
    .lower.toString()) endif) endif) endif;

entrypoint rule World() {
  do {
    thisModule.signature_text <- thisModule.signature_text
         + 'module model\n\n';
    thisModule.signature_text <- thisModule.signature_text
         + 'open world_structure[World]\n\n';
    thisModule.command_text <- 'run {}';
    thisModule.fact_text <- 'fact additional_facts {\n';
    thisModule.fact_text <- thisModule.fact_text + '  all
        w : World, x: (@next.w).domain_of_quantification |
        (x not in w.domain_of_quantification) => (x not in
        (( w. ^next).domain_of_quantification))\n';
    thisModule.fact_text <- thisModule.fact_text + '  all
        x: (' + thisModule.signature_names() + ') | some w:
         World | x in w.domain_of_quantification\n}\n';
    thisModule.text <- thisModule.final_text();
    thisModule.text.writeTo(thisModule.path);
    thisModule.debug('Module');
  }
}
```

```

rule SubstanceSortal2Signature {
    from
      s: OntoUML!SubstanceSortal
  using {
    abstract_keyword : String = if ((s.isAbstract = true) or thisModule.generalization_Sets_of_subKinds_subTypes_of(s)->exists(x | x.isCovering = true)) then 'abstract ' else '' endif;
  }
  do {
      thisModule.signature_text <- thisModule.signature_text + abstract_keyword + 'sig ' + s.name + ' {}\n';
    thisModule.text <- thisModule.final_text();
    thisModule.text.writeTo(thisModule.path);
    thisModule.debug('SubstanceSortal ' + s.name);
  }
}

rule SubKind2Subsignature {
    from
      s: OntoUML!SubKind
  using {
    abstract_keyword : String = if ((s.isAbstract = true) or thisModule.generalization_Sets_of_subKinds_subTypes_of(s)->exists(x | x.isCovering = true)) then 'abstract ' else '' endif;
    supertype_keyword : String = if (thisModule.generalization_Sets_of_superTypes_of(s)->exists(x | x.isDisjoint = true)) then ' extends ' else (if (thisModule.superTypes_set(s).size() > 0) then ' in ' else '' endif) endif;
    }
    do {
         thisModule.signature_text <- thisModule.
             signature_text + abstract_keyword + ’sig ’ + s.
             name + supertype_keyword + thisModule.
             superTypes_name_list(s) + ’ {}\n’;
       thisModule.text <- thisModule.final_text();
       thisModule.text.writeTo(thisModule.path);
       thisModule.debug(’SubKind ’ + s.name);
    }
}

rule Mode2Signature {
       from
         m: OntoUML!Mode
    using {
       abstract_keyword : String = if ((m.isAbstract = true)
           or thisModule.
           generalization_Sets_of_subKinds_subTypes_of(m)->
           exists(x | x.isCovering = true)) then ’abstract ’
           else ’’ endif;
       supertype_keyword : String = if (thisModule.
           generalization_Sets_of_superTypes_of(m)->exists(x |
            x.isDisjoint = true)) then ’ extends ’ else (if (
           thisModule.superTypes_set(m).size() > 0) then ’ in
           ’ else ’’ endif) endif;
       characterizations : OrderedSet(OntoUML!
           Characterization) = OntoUML!Characterization.
           allInstances()->select(x | x.source->exists(y | if
           y.oclIsKindOf(OntoUML!Property) then (thisModule.
           derive_endType(y) = m) else false endif));
       characterizations_str : String = characterizations->
           iterate(x; str : String = ’’ | str + ’   ’ + x.name
           + ’: ’+ thisModule.cardinality(x.target->any(y |
           true)) + ’ ’ + thisModule.allSuperTypes(thisModule.
```

```
      derive_endType(x.target->any(y | true)))->including
      (thisModule.derive_endType(x.target->any(y | true))
      )->select(x | x.oclIsKindOf(OntoUML!SubstanceSortal
      ))->any(x | true).name + ',\n');
  characterizations_cardinalities : String =
      characterizations->iterate(x; str : String = '' |
      str + thisModule.cardinality_characterization1(x.
      name,x.target->any(y | true)) + '\n');
  signature_constraints : String = let str: String =
      characterizations_cardinalities in (if (str <> '')
      then ('{\n' + str + '}') else '' endif);
    world_facts : String = let world_fact_str: String =
        ' all x: ' + m.name + ':>domain_of_quantification
         | ' + characterizations->iterate(x; str : String
         = '' | str + '(x.' + x.name + ' in ' +
        thisModule.derive_endType(x.target->any(y | true)
        ).name + ') and ') in (if (world_fact_str.size()
        > 0) then world_fact_str.substring(1,(
        world_fact_str.size()-5)) + '\n' else '' endif);
}
do {
  world_facts <- world_facts + characterizations->
      iterate(x; str : String = '' | str + thisModule.
      cardinality_characterization2(x.name,thisModule.
      name_side_target(x),x.source->any(y | true)) + '\n'
      );
    thisModule.signature_text <- thisModule.
        signature_text + abstract_keyword + 'sig ' + m.
        name + supertype_keyword + thisModule.
        superTypes_name_list(m) + ' {\n' +
        characterizations_str + '}' +
        signature_constraints + '\n';
  thisModule.signature_facts_world <- thisModule.
      signature_facts_world + world_facts;
  thisModule.text <- thisModule.final_text();
```

```
    thisModule.text.writeTo(thisModule.path);
    thisModule.debug('Mode ' + m.name);
  }
}

rule Relator2Signature {
    from
      r: OntoUML!Relator
  using {
    abstract_keyword : String = if ((r.isAbstract = true)
        or thisModule.
        generalization_Sets_of_subKinds_subTypes_of(r)->
        exists(x | x.isCovering = true)) then 'abstract '
        else '' endif;
    supertype_keyword : String = if (thisModule.
        generalization_Sets_of_superTypes_of(r)->exists(x |
         x.isDisjoint = true)) then ' extends ' else (if (
        thisModule.superTypes_set(r).size() > 0) then ' in
        ' else '' endif) endif;
    mediations : OrderedSet(OntoUML!Mediation) = OntoUML!
        Mediation.allInstances()->select(x | x.source->
        exists(y | if y.oclIsKindOf(OntoUML!Property) then
        (thisModule.derive_endType(y) = r) else false endif
        ));
    derivations : OrderedSet(OntoUML!Derivation) = OntoUML
        !Derivation.allInstances()->select(x | x.target->
        exists(y | if y.oclIsKindOf(OntoUML!Property) then
        (thisModule.derive_endType(y) = r) else false endif
        ));
    mediations_str : String = mediations->iterate(x; str :
         String = '' | str + '  ' + x.name + ': '+
        thisModule.cardinality(x.target->any(y | true)) + '
         ' + thisModule.allSuperTypes(thisModule.
        derive_endType(x.target->any(y | true)))->including
        (thisModule.derive_endType(x.target->any(y | true))
        )->select(x | x.oclIsKindOf(OntoUML!SubstanceSortal
        ))->any(x | true).name + ',\n');
    derivations_str : String = derivations->iterate(x; str
         : String = '' | let mediated_class1: OntoUML!
        ObjectClass = thisModule.derive_endType(thisModule.
        derive_endType(x.source->any(y | true)).
        associationEnd->at(1)) in let mediated_class2:
        OntoUML!ObjectClass = thisModule.derive_endType(
        thisModule.derive_endType(x.source->any(y | true)).
        associationEnd->at(2)) in let mediation1: OntoUML!
        Mediation = mediations->select(y | y.target->exists
        (z | if z.oclIsKindOf(OntoUML!Property) then (
        thisModule.derive_endType(z) = mediated_class1)
        else false endif))->at(1) in let mediation2:
        OntoUML!Mediation = mediations->select(y | y.target
        ->exists(z | if z.oclIsKindOf(OntoUML!Property)
        then (thisModule.derive_endType(z) =
        mediated_class2) else false endif))->at(1) in (str
        + ' ' + x.name + ': '+ mediation1.name + ' ' +
        thisModule.cardinality(mediation1.target->any(y |
        true)) + ' -> ' + thisModule.cardinality(mediation2
        .target->any(y | true)) + ' ' + mediation2.name + '
        ,\n'));
      mediations_generalization : String = thisModule.
          generalization_relational_constraint(mediations);
    derivations_generalization : String = thisModule.
        generalization_relational_constraint(derivations);
    mediations_cardinalities : String = mediations->
        iterate(x; str : String = '' | str + thisModule.
        cardinality_mediation1(x.name,x.target->any(y |
        true)) + '\n');
    derivations_cardinalities : String = derivations->
        iterate(x; str : String = '' | let mediated_class1:
         OntoUML!ObjectClass = thisModule.derive_endType(
        thisModule.derive_endType(x.source->any(y | true)).
        associationEnd->at(1)) in let mediated_class2:
        OntoUML!ObjectClass = thisModule.derive_endType(
        thisModule.derive_endType(x.source->any(y | true)).
        associationEnd->at(2)) in let mediation1: OntoUML!
        Mediation = mediations->select(y | y.target->exists
        (z | if z.oclIsKindOf(OntoUML!Property) then (
        thisModule.derive_endType(z) = mediated_class1)
        else false endif))->at(1) in let mediation2:
        OntoUML!Mediation = mediations->select(y | y.target
        ->exists(z | if z.oclIsKindOf(OntoUML!Property)
        then (thisModule.derive_endType(z) =
        mediated_class2) else false endif))->at(1) in (str
        + thisModule.cardinality_derivation1(x.name,
        mediation1.name,mediation2.target->any(y | true)) +
         '\n' + thisModule.cardinality_derivation2(x.name,
        mediation2.name,mediation1.target->any(y | true)) +
         '\n'));
    signature_constraints : String = let str: String =
        mediations_generalization +
        derivations_generalization +
        mediations_cardinalities +
        derivations_cardinalities in (if (str <> '') then (
        '{\n' + str + '}') else '' endif);
    world_facts : String = let world_fact_str: String = '
        all x: ' + r.name + ':>domain_of_quantification | '
         + mediations->iterate(x; str : String = '' | str +
         '(x.' + x.name + ' in ' + thisModule.
        derive_endType(x.target->any(y | true)).name + ')
        and ') in (if (world_fact_str.size() > 0) then
        world_fact_str.substring(1,(world_fact_str.size()
        -5)) + '\n' else '' endif);
  }
  do {
    world_facts <- world_facts + mediations->iterate(x;
        str : String = '' | str + thisModule.
        cardinality_mediation2(x.name,thisModule.
        name_side_target(x),x.source->any(y | true)) + '\n'
        );
    world_facts <- world_facts + derivations->iterate(x;
        str : String = '' | str + '  ' + thisModule.
        derive_endType(x.source->any(y | true)).name + ' =
        (' + r.name + ':>domain_of_quantification).' + x.
        name + '\n');
      thisModule.signature_text <- thisModule.
          signature_text + abstract_keyword + 'sig ' + r.
          name + supertype_keyword + thisModule.
          superTypes_name_list(r) + ' {\n' + mediations_str
           + derivations_str + '}' + signature_constraints
          + '\n';
    thisModule.signature_facts_world <- thisModule.
        signature_facts_world + world_facts;
    thisModule.text <- thisModule.final_text();
    thisModule.text.writeTo(thisModule.path);
    thisModule.debug('Relator ' + r.name);
  }
}

rule SimpleDatatype2Signature {
    from
      s: OntoUML!SimpleDatatype
  using {
    abstract_keyword : String = if ((s.isAbstract = true)
        or thisModule.
        generalization_Sets_of_subKinds_subTypes_of(s)->
        exists(x | x.isCovering = true)) then 'abstract '
        else '' endif;
    supertype_keyword : String = if (thisModule.
        generalization_Sets_of_superTypes_of(s)->exists(x |
         x.isDisjoint = true)) then ' extends ' else (if (
        thisModule.superTypes_set(s).size() > 0) then ' in
        ' else '' endif) endif;
  }
  do {
      thisModule.signature_text <- thisModule.
          signature_text + abstract_keyword + 'sig ' + s.
          name + supertype_keyword + thisModule.
          superTypes_name_list(s) + ' {}\n';
    thisModule.text <- thisModule.final_text();
    thisModule.text.writeTo(thisModule.path);
    thisModule.debug('SimpleDatatype ' + s.name);
  }
}

rule StructuralDatatype2Signature {
    from
      s: OntoUML!StructuralDatatype
  using {
    abstract_keyword : String = if ((s.isAbstract = true)
        or thisModule.
        generalization_Sets_of_subKinds_subTypes_of(s)->
        exists(x | x.isCovering = true)) then 'abstract '
        else '' endif;
    supertype_keyword : String = if (thisModule.
        generalization_Sets_of_superTypes_of(s)->exists(x |
         x.isDisjoint = true)) then ' extends ' else (if (
        thisModule.superTypes_set(s).size() > 0) then ' in
        ' else '' endif) endif;
    datatype_relationships : OrderedSet(OntoUML!
        DatatypeRelationship) = OntoUML!
        DatatypeRelationship.allInstances()->select(x | x.
        source->exists(y | if y.oclIsKindOf(OntoUML!
        Property) then (thisModule.derive_endType(y) = s)
        else false endif));
    datatype_relationships_str : String =
        datatype_relationships->iterate(x; str : String = '
      ' | str + '  ' + x.name + ': '+ thisModule.
      cardinality(x.target->any(y | true)) + ' ' +
      thisModule.derive_endType(x.target->any(y | true)).
      name + ',\n');
  datatype_relationships_cardinalities : String =
      datatype_relationships->iterate(x; str : String = '
      ' | str + thisModule.cardinality_mediation1(x.name,
      x.target->any(y | true)) + '\n');
  datatype_relationships_generalization : String =
      thisModule.generalization_relational_constraint(
      datatype_relationships);
  canonicity_fact : String = ' all x,y: ' + s.name + '
      | ' + (let canonicity_fact_str : String =
      datatype_relationships->iterate(x; str : String = '
      (' | str + '(x.@' + x.name + ' = ' + 'y.@' + x.name
       + ') and ') in canonicity_fact_str.substring(1,(
      canonicity_fact_str.size()-5))) + ') implies (x = y
      )\n';
  signature_constraints : String = let str: String =
      datatype_relationships_generalization +
      datatype_relationships_cardinalities +
      canonicity_fact in (if (str <> '') then ('{\n' +
      str + '}') else '' endif);
}
do {
    thisModule.signature_text <- thisModule.
        signature_text + abstract_keyword + 'sig ' + s.
        name + supertype_keyword + thisModule.
        superTypes_name_list(s) + ' {\n' +
        datatype_relationships_str + '}' +
        signature_constraints + '\n';
  thisModule.text <- thisModule.final_text();
  thisModule.text.writeTo(thisModule.path);
  thisModule.debug('StructuredDatatype ' + s.name);
}
```

```
}

rule Category2Function {
    from
      s: OntoUML!Category
  do {
      if(thisModule.subTypes_set(s)->forAll(x | x.
          oclIsKindOf(OntoUML!Category)))
        thisModule.function_text <- thisModule.
            function_text + 'fun ' + s.name + ': univ {' +
            thisModule.subTypes_name_union_set(s) + '}\n';
      else
          thisModule.function_text <- thisModule.
              function_text + 'fun ' + s.name + ': (' +
              thisModule.subTypes_name_union_set(s) + ')' +
               ' {\n  ' + thisModule.
              subTypes_name_union_set(s) + '\n}\n';
    thisModule.text <- thisModule.final_text();
    thisModule.text.writeTo(thisModule.path);
    thisModule.debug('Category ' + s.name);
  }
}

rule Phase2Relation {
    from
      p: OntoUML!Phase
  do {
    if (OntoUML!GeneralizationSet.allInstances()->select(x
         | if (x.generalization.size() > 0) then (if (x.
        generalization->exists(y | thisModule.
        derive_specific(y) = p)) then true else false endif
        ) else false endif)->isEmpty()) {
        thisModule.relation_phases <- thisModule.
            relation_phases + '   ' + p.name + ': set ' +
            thisModule.superTypes_name_union_set(p) + ':>
            domain_of_quantification,\n';
    }
    thisModule.signature_facts_world <- thisModule.
        signature_facts_world + ' all x: ' + thisModule.
        allSuperTypes(p)->select(x | x.oclIsKindOf(OntoUML!
        SubstanceSortal))->any(x | true).name + ' | some w:
         World | x in w.@' + p.name + '\n';
    thisModule.text <- thisModule.final_text();
    thisModule.text.writeTo(thisModule.path);
    thisModule.debug('Phase ' + p.name);
  }
}

rule Role2Relation {
    from
      p: OntoUML!Role
  do {
    if (thisModule.superTypes_set(p)->forAll(x | not x.
        oclIsKindOf(OntoUML!AntiRigidSortalClass))) {
      thisModule.signature_facts_world <- thisModule.
          signature_facts_world + ' all x: ' + p.name + '
          | some w: World | (x in w.
          @domain_of_quantification) and (x not in w.@' + p
          .name + ')\n';
    }
    else {
        thisModule.signature_facts_world <- thisModule.
            signature_facts_world + '-- all x: ' + p.name
            + ' | some w: World | (x in w.
            @domain_of_quantification) and (x not in w.@' +
             p.name + ')\n';
    }
    thisModule.text <- thisModule.final_text();
    thisModule.text.writeTo(thisModule.path);
    thisModule.debug('Role ' + p.name);
  }
}

rule Mixin2Relation {
    from
      m: OntoUML!Mixin
  do {
    thisModule.signature_facts_world <- thisModule.signature_facts_world + ' all x: (' + thisModule.subTypes_name_union_set_special_rigid_sortal_subtypes(m) + ') | x in ' + m.name + '\n';
      thisModule.text <- thisModule.final_text();
    thisModule.text.writeTo(thisModule.path);
    thisModule.debug('Mixin ' + m.name);
  }
}

rule RoleMixin2Relation {
    from
      r: OntoUML!RoleMixin
  do {
    thisModule.signature_facts_world <- thisModule.signature_facts_world + ' all x: (' + thisModule.subTypes_name_union_set(r) + ') | x in ' + r.name + '\n';
      thisModule.text <- thisModule.final_text();
    thisModule.text.writeTo(thisModule.path);
    thisModule.debug('RoleMixin ' + r.name);
  }
}

rule Meronymic2Relation {
    from
      m: OntoUML!Meronymic
    using {
            source_min_cardinality : Integer = m.source->any
                (x | true).lower;
            source_max_cardinality : Integer = m.source->any
                (x | true).upper;
            target_min_cardinality : Integer = m.target->any
                (x | true).lower;
            target_max_cardinality : Integer = m.target->any
                (x | true).upper;
            cardinality_source : String = '';
            cardinality_target : String = '';
        cardinality_fact_source : String = thisModule.
            cardinality_fact1(m.name,thisModule.
            name_side_source(m),m.target->any(x | true));
        cardinality_fact_target : String = thisModule.
            cardinality_fact2(m.name,thisModule.
            name_side_target(m),m.source->any(x | true));
        tlrss : OrderedSet (OntoUML!Classifier) = thisModule
            .top_level_rigid_sortals_connected_on_source(m);
        tlrst : OrderedSet (OntoUML!Classifier) = thisModule
            .top_level_rigid_sortals_connected_on_target(m);
        world_facts : String = '';
        }
    do {
      if ((source_min_cardinality = 0) and (
          source_max_cardinality = 1)) {
          cardinality_source <- 'lone';
      }
      else {
          if ((source_min_cardinality = 1) and (
              source_max_cardinality = 1)) {
              cardinality_source <- 'one';
          }
          else {
              if ((source_min_cardinality = 1) and (
                  source_max_cardinality = 0-1)) {
```

```
                cardinality_source <- 'some';
            }
            else {
                cardinality_source <- 'set';
            }
        }
    }
    if ((target_min_cardinality = 0) and (
        target_max_cardinality = 1)) {
        cardinality_target <- 'lone';
    }
    else {
        if ((target_min_cardinality = 1) and (
            target_max_cardinality = 1)) {
            cardinality_target <- 'one';
        }
        else {
            if ((target_min_cardinality = 1) and (
                target_max_cardinality = 0-1)) {
                cardinality_target <- 'some';
            }
            else {
                cardinality_target <- 'set';
            }
        }
    }
      thisModule.relation_relations <- thisModule.
          relation_relations + '  ' + m.name + ': set ' +
          thisModule.derive_endType(m.target->any(x | true)
          ).name + (if (thisModule.derive_endType(m.target
          ->any(x | true)).oclIsKindOf(OntoUML!
          SubstanceSortal)) then ':>
          domain_of_quantification' else '' endif) + ' ' +
          cardinality_target + ' -> ' + cardinality_source
          + ' ' + thisModule.derive_endType(m.source->any(x
                | true)).name + (if (thisModule.derive_endType(m
                .source->any(x | true)).oclIsKindOf(OntoUML!
                SubstanceSortal)) then ’:>
                domain_of_quantification’ else ’’ endif) + ’,\n’;
            if ((m.isImmutablePart = true) or (m.isEssential =
                true) or (m.target->any(x | true).isReadOnly =
                true)) {
              world_facts <- world_facts + if ((tlrss.size() >
                  0) and (tlrst.size() > 0)) then (’ all x: ’ +
                  tlrss->any(x | true).name + ’, w0, w1: (@’ + m.
                  name + ’.x).’ + tlrst->any(x | true).name + ’ |
                   (w0.@’ + m.name + ’).x = (w1.@’ + m.name + ’).
                  x  -- essential or immutablePart or readOnly
                  target.\n’) else ’’ endif;
            }
            if ((m.isImmutableWhole = true) or (m.isInseparable
                = true) or (m.source->any(x | true).isReadOnly =
                true)) {
              world_facts <- world_facts + if ((tlrss.size() >
                  0) and (tlrst.size() > 0)) then (’ all x: ’ +
                  tlrst->any(x | true).name + ’, w0, w1: (@’ + m.
                  name + ’.’ + tlrss->any(x | true).name + ’).x |
                   x.(w0.@’ + m.name + ’) = x.(w1.@’ + m.name + ’
                  ) -- inseparable or immutableWhole or readOnly
                  source.\n’) else ’’ endif;
            }
        world_facts <- world_facts + (if (
            cardinality_fact_source <> ’’) then (
            cardinality_fact_source + ’\n’) else ’’ endif) + (
            if (cardinality_fact_target <> ’’) then (
            cardinality_fact_target + ’\n’) else ’’ endif);
        thisModule.signature_facts_world <- thisModule.
            signature_facts_world + world_facts;
        thisModule.text <- thisModule.final_text();
        thisModule.text.writeTo(thisModule.path);
```

```
    thisModule.debug('Meronymic ' + m.name);
  }
}

rule DatatypeRelationship2Relation {
    from
      d: OntoUML!DatatypeRelationship (
          not d.source->exists(x | if x.oclIsKindOf(
              OntoUML!Property) then (thisModule.
              derive_endType(x).oclIsKindOf(OntoUML!
              StructuralDatatype)) else false endif)
      )
    using {
          source_min_cardinality : Integer = d.source->any
              (x | true).lower;
          source_max_cardinality : Integer = d.source->any
              (x | true).upper;
          target_min_cardinality : Integer = d.target->any
              (x | true).lower;
          target_max_cardinality : Integer = d.target->any
              (x | true).upper;
      name : String = d.target->any(x | true).name;
          cardinality_source : String = '';
          cardinality_target : String = '';
      cardinality_fact_source : String = thisModule.
          cardinality_fact1(name,thisModule.
          name_side_source(d),d.target->any(x | true));
      cardinality_fact_target : String = thisModule.
          cardinality_fact2(name,thisModule.
          name_side_target(d),d.source->any(x | true));
      tlrss : OrderedSet (OntoUML!Classifier) = thisModule
          .top_level_rigid_sortals_connected_on_source(d);
      tlrst : OrderedSet (OntoUML!Classifier) = thisModule
          .top_level_rigid_sortals_connected_on_target(d);
      world_facts : String = '';
    }
do {
  if ((source_min_cardinality = 0) and (
      source_max_cardinality = 1)) {
      cardinality_source <- 'lone';
  }
  else {
      if ((source_min_cardinality = 1) and (
          source_max_cardinality = 1)) {
          cardinality_source <- 'one';
      }
      else {
          if ((source_min_cardinality = 1) and (
              source_max_cardinality = 0-1)) {
              cardinality_source <- 'some';
          }
          else {
              cardinality_source <- 'set';
          }
      }
  }
  if ((target_min_cardinality = 0) and (
      target_max_cardinality = 1)) {
      cardinality_target <- 'lone';
  }
  else {
      if ((target_min_cardinality = 1) and (
          target_max_cardinality = 1)) {
          cardinality_target <- 'one';
      }
      else {
          if ((target_min_cardinality = 1) and (
              target_max_cardinality = 0-1)) {
              cardinality_target <- 'some';
          }
```

```
        else {
            cardinality_target <- ’set’;
        }
    }
}
  thisModule.relation_relations <- thisModule.
      relation_relations + ’  ’ + name + ’: set ’ +
      thisModule.derive_endType(d.target->any(x | true)
      ).name + (if (thisModule.derive_endType(d.target
      ->any(x | true)).oclIsKindOf(OntoUML!
      SubstanceSortal)) then ’:>
      domain_of_quantification’ else ’’ endif) + ’ ’ +
      cardinality_target + ’ -> ’ + cardinality_source
      + ’ ’ + thisModule.derive_endType(d.source->any(x
       | true)).name + (if (thisModule.derive_endType(d
      .source->any(x | true)).oclIsKindOf(OntoUML!
      SubstanceSortal)) then ’:>
      domain_of_quantification’ else ’’ endif) + ’,\n’;
  if (d.target->any(x | true).isReadOnly = true) {
    world_facts <- world_facts + if ((tlrss.size() >
        0) and (tlrst.size() > 0)) then (’ all x: ’ +
        tlrss->any(x | true).name + ’, w0, w1: (@’ +
        name + ’.x).’ + tlrst->any(x | true).name + ’ |
         (w0.@’ + name + ’).x = (w1.@’ + name + ’).x
        -- readOnly target.\n’) else ’’ endif;
  }
  if (d.source->any(x | true).isReadOnly = true) {
    world_facts <- world_facts + if ((tlrss.size() >
        0) and (tlrst.size() > 0)) then (’ all x: ’ +
        tlrst->any(x | true).name + ’, w0, w1: (@’ +
        name + ’.’ + tlrss->any(x | true).name + ’).x |
         x.(w0.@’ + name + ’) = x.(w1.@’ + name + ’) --
         readOnly source.\n’) else ’’ endif;
  }
world_facts <- world_facts + (if (
        cardinality_fact_source <> '') then (
        cardinality_fact_source + '\n') else '' endif) + (
        if (cardinality_fact_target <> '') then (
        cardinality_fact_target + '\n') else '' endif);
    thisModule.signature_facts_world <- thisModule.
        signature_facts_world + world_facts;
    thisModule.text <- thisModule.final_text();
    thisModule.text.writeTo(thisModule.path);
    thisModule.debug('DatatypeRelationship ' + name);
  }
}

rule Association2Relation {
    from
      m: OntoUML!Association
    using {
          associationEnd1_min_cardinality : Integer = m.
              associationEnd->at(1).lower;
          associationEnd1_max_cardinality : Integer = m.
              associationEnd->at(1).upper;
          associationEnd2_min_cardinality : Integer = m.
              associationEnd->at(2).lower;
          associationEnd2_max_cardinality : Integer = m.
              associationEnd->at(2).upper;
          cardinality_associationEnd1 : String = '';
          cardinality_associationEnd2 : String = '';
      cardinality_fact_associationEnd1 : String =
          thisModule.cardinality_fact2(m.name,thisModule.
          name_side_associationEnd(m,1),m.associationEnd->
          at(2));
      cardinality_fact_associationEnd2 : String =
          thisModule.cardinality_fact1(m.name,thisModule.
          name_side_associationEnd(m,2),m.associationEnd->
          at(1));
      tlrsa1 : OrderedSet (OntoUML!Classifier) =
        thisModule.
        top_level_rigid_sortals_connected_on_associationEnd
        (m,1);
    tlrsa2 : OrderedSet (OntoUML!Classifier) =
        thisModule.
        top_level_rigid_sortals_connected_on_associationEnd
        (m,2);
    world_facts : String = '';
    }
do {
  if ((associationEnd1_min_cardinality = 0) and (
      associationEnd1_max_cardinality = 1)) {
      cardinality_associationEnd1 <- 'lone';
  }
  else {
      if ((associationEnd1_min_cardinality = 1) and (
          associationEnd1_max_cardinality = 1)) {
          cardinality_associationEnd1 <- 'one';
      }
      else {
          if ((associationEnd1_min_cardinality = 1) and
              (associationEnd1_max_cardinality = 0-1)) {
              cardinality_associationEnd1 <- 'some';
          }
          else {
              cardinality_associationEnd1 <- 'set';
          }
      }
  }
  if ((associationEnd2_min_cardinality = 0) and (
      associationEnd2_max_cardinality = 1)) {
      cardinality_associationEnd2 <- 'lone';
  }
  else {
      if ((associationEnd2_min_cardinality = 1) and (
            associationEnd2_max_cardinality = 1)) {
            cardinality_associationEnd2 <- 'one';
        }
        else {
            if ((associationEnd2_min_cardinality = 1) and
                (associationEnd2_max_cardinality = 0-1)) {
                cardinality_associationEnd2 <- 'some';
            }
            else {
                cardinality_associationEnd2 <- 'set';
            }
        }
    }
      if (m.associationEnd->at(2).isReadOnly = true) {
        world_facts <- world_facts + if ((tlrsa1.size() >
            0) and (tlrsa2.size() > 0)) then (' all x: ' +
            tlrsa1->any(x | true).name + ', w0, w1: (@' + m
            .name + '.x).' + tlrsa2->any(x | true).name + '
             | (w0.@' + m.name + ').x = (w1.@' + m.name + '
            ).x  -- readOnly target.\n') else '' endif;
      }
      if (m.associationEnd->at(1).isReadOnly = true) {
        world_facts <- world_facts + if ((tlrsa1.size() >
            0) and (tlrsa2.size() > 0)) then (' all x: ' +
            tlrsa2->any(x | true).name + ', w0, w1: (@' + m
            .name + '.' + tlrsa1->any(x | true).name + ').x
             | x.(w0.@' + m.name + ') = x.(w1.@' + m.name +
             ') -- readOnly source.\n') else '' endif;
      }
      thisModule.relation_relations <- thisModule.
          relation_relations + '  ' + m.name + ': set ' +
          thisModule.derive_endType(m.associationEnd->at(1)
          ).name + (if (thisModule.derive_endType(m.
          associationEnd->at(1)).oclIsKindOf(OntoUML!
          SubstanceSortal)) then ':>
          domain_of_quantification’ else ’’ endif) + ’ ’ +
          cardinality_associationEnd2 + ’ -> ’ +
          cardinality_associationEnd1 + ’ ’ + thisModule.
          derive_endType(m.associationEnd->at(2)).name + (
          if (thisModule.derive_endType(m.associationEnd->
          at(2)).oclIsKindOf(OntoUML!SubstanceSortal)) then
           ’:>domain_of_quantification’ else ’’ endif) + ’
          ,\n’;
    world_facts <- world_facts + (if (
        cardinality_fact_associationEnd1 <> ’’) then (
        cardinality_fact_associationEnd1 + ’\n’) else ’’
        endif) + (if (cardinality_fact_associationEnd2 <> ’
        ’) then (cardinality_fact_associationEnd2 + ’\n’)
        else ’’ endif);
    thisModule.signature_facts_world <- thisModule.
        signature_facts_world + world_facts;
    thisModule.text <- thisModule.final_text();
    thisModule.text.writeTo(thisModule.path);
    thisModule.debug(’Association ’ + m.name);
  }
}

rule GeneralizationSet2Fact {
    from
      g : OntoUML!GeneralizationSet (
        (g.isDisjoint or g.isCovering) -- Otherwise, this
            fact would have no body.
        and not g.generalization->collect(x | thisModule.
            derive_specific(x))->forAll(x | x.oclIsKindOf(
            OntoUML!SubKind))
      )
    using {
      general : OntoUML!Element = g.generalization->
          collect(x | thisModule.derive_general(x))->any(x
          | true);
      specifics : OrderedSet(OntoUML!Element) = g.generalization->collect(x | thisModule.derive_specific(x));
      facts : String = '';
    }
    do {
        if (specifics->forAll(x | x.oclIsKindOf(OntoUML!Relationship))) { -- If the generalizations are between associations.
            if ((g.isDisjoint = true) and (specifics->size() >= 2)) {
                facts <- '  disj[' + thisModule.names_list(specifics) + ']\n';
                if (g.isCovering = true)
                  facts <- facts + '   ' + general.name + ' = ' + thisModule.names_union_set(specifics) + '\n';
            }
            else {
                if (g.isCovering = true)
          facts <- facts + '   ' + general.name + ' = ' + thisModule.names_union_set(specifics) + '\n';
            }
        }
      else { -- If the generalizations are between classes.
            if ((g.isDisjoint = true) and (specifics->size() >= 2)) {
                facts <- '  disj[' + thisModule.names_list(specifics) + ']\n';
                if (g.isCovering = true)
                    if (general.oclIsKindOf(OntoUML!SubstanceSortal) or general.oclIsKindOf(OntoUML!Relator))
                    facts <- facts + '   ' + general.name +
                 ':> domain_of_quantification = ' + thisModule.names_union_set(specifics) + '\n';
          else
            facts <- facts + '  ' + general.name + ' = ' + thisModule.names_union_set(specifics) + '\n';
      }
      else {
        if (g.isCovering = true)
          if (general.oclIsKindOf(OntoUML!SubstanceSortal) or general.oclIsKindOf(OntoUML!Relator))
          facts <- '  ' + general.name + ':> domain_of_quantification = ' + thisModule.names_union_set(specifics) + '\n';
          else
            facts <- '  ' + general.name + ' = ' + thisModule.names_union_set(specifics) + '\n';
      }
   }
  thisModule.signature_facts_world <- thisModule.signature_facts_world + facts;
  thisModule.text <- thisModule.final_text();
  thisModule.text.writeTo(thisModule.path);
  thisModule.debug('GeneralizationSet ' + g.name);
  }
}
```

APPENDIX C – OntoUML Editor Manual

This appendix explains how to install and execute the OntoUML Editor. Notice that a working Java Runtime Environment (JRE) is a pre-requisite.

In order to install the OntoUML Editor and create an OntoUML model, you must follow the steps show in 86, which, at the writing of this thesis, are the steps shown in the next sections.

C.1 Installing the OntoUML Editor

There are two ways of using the OntoUML Editor: (i) as a standalone product, or (ii) as an Eclipse plug-in. Subsection C.1.1 explains how to obtain the standalone binaries, while subsection C.1.2 deals with the installation of the OntoUML Editor as an Eclipse plug-in.

C.1.1 Installing the Standalone OntoUML Editor

In the Downloads tab of the OntoUML Editor's site[1], there are binaries for different combinations of Operational Systems (OSs) (*viz.*: Advanced Interactive eXecutive[2] (AIX), Hewlett Packard UniX[3] (HP-UX), Linux, Mac OS X, Solaris and Windows), processor types (*viz.*: IA64-32, Performance Optimization With Enhanced RISC - Performance Computing (PowerPC), Scalable Processor Architecture[4] (SPARC), x86 and x86-64) and widget toolkits (*viz.*: Carbon, GIMP Toolkit[5] (GTK), Motif, Win32 Application Programming Interface (API) and Windows Presentation Foundation[6] (WPF)). You have to choose a suitable combination. The options are:

- OS: AIX

[1] http://code.google.com/p/ontouml/downloads/list.
[2] http://www-03.ibm.com/systems/power/software/aix.
[3] http://www.hp.com/go/hpux.
[4] http://www.sparc.org.
[5] http://www.gtk.org.
[6] http://windowsclient.net/wpf.

    - Processor type: PowerPC
        - Widget toolkit: Motif
- OS: HP-UX
    - Processor type: IA64-32
        - Widget toolkit: Motif
- OS: Linux
    - Processor type: PowerPC
        - Widget toolkit: GTK
    - Processor type: x86
        - Widget toolkit: GTK
    - Processor type: x86-64
        - Widget toolkit: GTK
- OS: Mac OS X
    - Processor type: PowerPC
        - Widget toolkit: Carbon
    - Processor type: x86
        - Widget toolkit: Carbon
- OS: Solaris
    - Processor type: SPARC
        - Widget toolkit: GTK
- OS: Windows NT and Windows 95 up to Windows 7 (The versions of Windows that use the Win32 API)
    - Processor type: x86
        - Widget toolkit: Win32 API
        - Widget toolkit: WPF
    - Processor type: x86-64
        - Widget toolkit: Win32 API

If you have no idea of what are those options, just download OntoUMLEditor1.2.0.win32.win32.x86.zip (you are probably using a 32 bit Windows).

After downloading, unzip the file, enter in the folder “eclipse” and run OntoUMLEditor[.exe].

C.1.2 Installing the OntoUML Editor Eclipse Plug-in

In order to install the OntoUML Editor Eclipse plug-in, you must follow the following steps:

1. Download the Eclipse Modeling Tools package for your OS at http://www.eclipse.org/downloads and unzip it (in case you do not have a working Eclipse);

2. If you have manually installed a previous version of OntoUML Editor in the “dropins” Eclipse folder, just delete the OntoUML Editor files from “dropins”;

3. Installing:

   - Installing from the repository (Recommended):

     - In Eclipse 3.5.x or 3.6.x (attention, the OntoUML Editor still does not work with these versions of Eclipse!), go to Help → Install New Software… → Add…, then put “OntoUML Editor” (without quotes) in the “Name” field and “http://ontouml.googlecode.com/svn/trunk/OntoUMLUpdateSite” (without quotes) in the “Location” field and then click OK. Now, deselect the button "Group items by catgory", select “OntoUML Editor” and click on Next → Next → Read and accept the terms of the license → Finish. If Eclipse complains about unsigned content, just click OK.

     - In Eclipse 3.4.2[7], go to Help → Software Updates → Available Software → Add Site and add the address http://ontouml.googlecode.com/svn/trunk/OntoUMLUpdateSite and click on OK. After, click on “Manage Sites”, select this Uniform Resource Locator (URL) or “OntoUML Editor Update Site” (if it was not selected by default) and click on OK. Click on Refresh. Now, open the repository “OntoUML Editor Update Site”, select OntoUML Editor and click on Install.

   - Installing manually (warning: if you install manually, you will not be notified of updates): Download the latest zip file from the Downloads tab and unzip it in the “dropins” folder of your Eclipse.

[7] One can get this version at http://www.eclipse.org/downloads/packages/eclipse-modeling-tools-includes-incubating-components/ganymedesr2.

4. Restart Eclipse when the installation finishes;
5. It is recommended to periodically make a search for updates (only works if installed from the repository): Go to Help → Software Updates → Installed Software and click on Update. You should make a backup of your models before opening them with a new version of the editor.

C.2 Creating an OntoUML Model

The creation of an OntoUML model in the standalone OntoUML Editor is different from the creation in the OntoUML Editor Eclipse plug-in. Subsection C.2.1 explains how to create an OntoUML model in the standalone OntoUML Editor, while subsection C.2.2 deals with the creation of OntoUML models in the OntoUML Editor Eclipse plug-in.

C.2.1 Creating an OntoUML Model in the Standalone OntoUML Editor

1. Execute the standalone OntoUML Editor;
2. Click on File → New → OntoUML Diagram → Name the file → Next → Finish;
3. You are ready to create an OntoUML model.

C.2.2 Creating an OntoUML Model in the OntoUML Editor Eclipse Plug-in

1. Execute Eclipse;
2. Create a project (if you do not have one): Click in File → New → Project… → General → Project → Next → Name it → Next → Next → Finish;
3. Create an OntoUML diagram: Right click on the folder just created → New → Example → OntoUML Diagram → Next → Give a name for the file → Next → Next → Finish;
4. You are ready to create an OntoUML model.

C.3 Tips

Avoid using the functions “cut”, “copy” or “paste” in OntoUML Editor, because they may corrupt your models. Instead, use the function “duplicate”, which is very similar. There are two ways of using “duplicate”:

- Select the classes and relations by pressing *Ctrl* and when you have selected the last element, keep *Ctrl* pressed and drag and drop the selection (using left mouse button) in a suitable place for the new elements; or
- Select the classes and relations by pressing *Ctrl* → right click in one of the selected elements → Edit → Duplicate → drag and drop the new elements in a suitable place.

This tutorial (or a newer version) is also available at 87.

APPENDIX D – OntoUML to Alloy ATL Transformation Manual

This appendix explains how to install and execute the ATL transformation from OntoUML to Alloy.

In order to install the ATL transformation from OntoUML to Alloy and create an OntoUML model, you must follow the steps show in 88, which, at the writing of this thesis, are the steps shown in the next sections.

D.1 Installing the Transformation

In order to install the ATL transformation from OntoUML to Alloy, you must follow the following steps:

1. Download the version 3.4.2 of Eclipse Modeling Tools package for your OS[1] and unzip it (in case you do not have a working Eclipse);
2. Execute Eclipse;
3. Create an ATL project (if you do not have one): Click in File → New → Project… → ATL → ATL Project → Next → in "Name" put "OntoUML2Alloy" (without quotes) → Finish → Click in "Yes" in the "Open Associated Perspective" window;
4. Open your file manager and find the folder of your just created OntoUML2Alloy ATL project. Then, download the files that are at http://ontouml.googlecode.com/svn/trunk/OntoUML2Alloy to this folder;
5. Download the latest version of Alloy Analyzer from http://alloy.mit.edu/community/software to your OntoUML2Alloy project folder;

[1] One can get this version at http://www.eclipse.org/downloads/packages/eclipse-modeling-tools-includes-incubating-components/ganymedesr2.

D.2 Transforming an OntoUML Model into an Alloy Specification

In order to transform an OntoUML model into an Alloy specification, you must follow the following steps:

1. Execute Eclipse;
2. Click on Run → Run Configurations… → ATL Transformation → New launch configuration → In “Name” put “OntoUML2Alloy” (without quotes), inside “Project”, in “Name” choose “OntoUML2Alloy” and in “ATL file”, choose “/OntoUML2Alloy/ OntoUML2Alloy.atl”. Within “Metamodels”, click on the “Workspace…” button, open “OntoUML2Alloy” and select “OntoUML.ecore”. Within “Source Models”, click on the “Workspace…” button and search and open your OntoUML model (the file with the “.ontouml” extension and not the “.ontouml_diagram” one). Within “Target Models”, put “/OntoUML2Alloy/you_can_delete_this_file” (without quotes) in the “Out:” field;
3. Click on the “Common” tab and select “Debug” and “Run” within “Display in favorites menu”;
4. Click in “Apply” → “Close”;
5. Click in the right side of the “Run” button → “OntoUML2Alloy”. The OntoUML2Alloy transformation will generate a file named “specification.als”;
6. Open your file manager and find the folder of your OntoUML2Alloy ATL project. Then execute the file “alloy4.jar” (the Alloy Analyzer). Within Alloy Analyzer, click in “Open” and find the file “specification.als”. Then click in “Options” → “Visualize Automatically”. Finally, click in “Execute” → “Run run$1”;
7. You can optionally use our predefined theme to visualize the generated Kipke structure: Within the window of the generated figure, click in “Theme” → “Load Theme…” → Select “alloy_analyzer_theme_world_structure.thm” → “Open”.

APPENDIX E – Alloy Analyzer Themes

In this appendix, we show the visualization themes that we created for customizing the visualization of the instances generated by the Alloy Analyzer. The theme shown in Listing 117 is to customize the visualization of the temporal ordering of worlds, as explained in subsection 5.6.1; and the theme shown in Listing 118 is to customize the visualization of the atoms by projecting them in worlds. These themes are licensed under GPLv3 (see annex A).

Listing 117: Theme for visualization of the temporal ordering of worlds.

```
<?xml version="1.0"?>
<alloy>

<view>

<defaultnode visible="no"/>

<defaultedge visible="no"/>

<node>
   <type name="Int"/>
   <type name="String"/>
   <type name="univ"/>
   <type name="seq/Int"/>
</node>

<node label="TemporalWorld">
   <type name="world_structure/TemporalWorld"/>
</node>

<node style="Bold" label="CurrentWorld">
```

```
   <type name="world_structure/CurrentWorld"/>
</node>

<node style="Dashed" label="FutureWorld">
   <type name="world_structure/FutureWorld"/>
</node>

<node style="Dotted" label="CounterfactualWorld">
   <type name="world_structure/CounterfactualWorld"/>
</node>

<node style="Solid" label="PastWorld">
   <type name="world_structure/PastWorld"/>
</node>

<node visible="yes" color="White">
   <type name="World"/>
</node>

<edge color="Black" visible="yes" layout="yes">
   <relation name="next"> <type name="World"/> <type name="World"/> </relation>
</edge>

</view>

</alloy>
```

Listing 118: Theme for visualization of atoms projected in worlds.

```
<?xml version="1.0"?>
<alloy>

<view>

<projection> <type name="World"/> </projection>

```

```
<defaultnode visible="no" color="White"/>

<defaultedge color="Black"/>

<node>
   <type name="Int"/>
   <type name="Man"/>
   <type name="Organization"/>
   <type name="String"/>
   <type name="univ"/>
   <type name="Woman"/>
   <type name="World"/>
   <type name="seq/Int"/>
   <type name="world_structure/CounterfactualWorld"/>
   <type name="world_structure/CurrentWorld"/>
   <type name="world_structure/FutureWorld"/>
   <type name="world_structure/PastWorld"/>
   <type name="world_structure/TemporalWorld"/>
   <set name="ActiveOrganization" type="Organization"/>
   <set name="Adult" type="Person"/>
   <set name="Child" type="Person"/>
   <set name="DeceasedPerson" type="Person"/>
   <set name="ExtinctOrganization" type="Organization"/>
   <set name="FunctionalBrain" type="Brain"/>
   <set name="FunctionalHeart" type="Heart"/>
   <set name="LivingPerson" type="Person"/>
   <set name="NonfunctionalBrain" type="Brain"/>
   <set name="NonfunctionalHeart" type="Heart"/>
   <set name="School" type="Organization"/>
   <set name="Student" type="Person"/>
   <set name="Teenager" type="Person"/>
</node>

<node label="BiologicalOrgan">
   <set name="$BiologicalOrgan" type="Brain"/>
   <set name="$BiologicalOrgan" type="Heart"/>
</node>

<node shape="Ellipse">
   <type name="Person"/>
</node>

<node shape="Hexagon">
   <type name="Enrollment"/>
</node>

<node shape="Trapezoid">
   <type name="Brain"/>
   <type name="Heart"/>
</node>

<node showlabel="no">
   <set name="$w" type="Organization"/>
   <set name="$w&apos;" type="Brain"/>
   <set name="$w&apos;&apos;" type="Person"/>
   <set name="$w&apos;&apos;&apos;" type="Brain"/>
   <set name="$w&apos;&apos;&apos;&apos;" type="Heart"/>
   <set name="$w&apos;&apos;&apos;&apos;&apos;" type="Person"/>
   <set name="$w&apos;&apos;&apos;&apos;&apos;&apos;" type="Heart"/>
   <set name="$w&apos;&apos;&apos;&apos;&apos;&apos;&apos;" type="Person"/>
   <set name="$w&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;" type="Organization"/>
   <set name="$w&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;" type="Person"/>
   <set name="$w&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;" type="Person"/>
```

```
  <set name="$w&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;" type="Brain"/>
  <set name="$w&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;" type="Enrollment"/>
  <set name="$w&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;" type="Heart"/>
  <set name="$w&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;" type="Organization"/>
  <set name="$w&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;&apos;" type="Person"/>
</node>

<node visible="yes" showlabel="no">
  <set name="domain_of_quantification" type="Brain"/>
  <set name="domain_of_quantification" type="Enrollment"/>
  <set name="domain_of_quantification" type="Heart"/>
  <set name="domain_of_quantification" type="Organization"/>
  <set name="domain_of_quantification" type="Person"/>
</node>

<edge visible="no" constraint="no">
  <relation name="derived_study"> <type name="Enrollment"/> <type name="Person"/> <type name="Organization"/> </relation>
</edge>

</view>

</alloy>
```

ANNEX A – GPLv3

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

   A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

   The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   (a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

   (b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

   (c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

   (d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display

Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an “aggregate” if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation’s users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

   (a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

   (b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

   (c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

   (d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

(e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

   "Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

   When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

   Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

   (a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

   (b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

   (c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

(d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

(e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

(f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

   You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor’s essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

```
<one line to give the program’s name and a brief idea of what it does.>

Copyright (C) <textyear>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>

This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.

ANNEX B – EPLv1

Eclipse Public License -v 1.0

THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS ECLIPSE PUBLIC LICENSE (“AGREEMENT”). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT’S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS

“Contribution” means:

a) in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and

b) in the case of each subsequent Contributor:

i) changes to the Program, and

ii) additions to the Program;

where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution ’originates’ from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor’s behalf. Contributions do not include additions to the Program which: (i) are separate modules of software distributed in conjunction with the Program under their own license agreement, and (ii) are not derivative works of the Program.

“Contributor” means any person or entity that distributes the Program.

“Licensed Patents” mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.

“Program” means the Contributions distributed in accordance with this Agreement.

“Recipient” means anyone who receives the Program under this Agreement, including all

Contributors.

2. GRANT OF RIGHTS

a) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Contribution of such Contributor, if any, and such derivative works, in source code and object code form.

b) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.

c) Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.

d) Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.

3. REQUIREMENTS

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

a) it complies with the terms and conditions of this Agreement; and

b) its license agreement:

i) effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;

ii) effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;

iii) states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and

iv) states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

a) it must be made available under this Agreement; and

b) a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.

4. COMMERCIAL DISTRIBUTION

Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor (“Commercial Contributor”) hereby agrees to defend and indemnify every other Contributor (“Indemnified Contributor”) against any losses, damages and costs (collectively “Losses”) arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

5. NO WARRANTY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement , including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

6. DISCLAIMER OF LIABILITY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. GENERAL

If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

If Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions of this Agreement and does not cure such failure in a reasonable period of time after becoming aware of such noncompliance. If all Recipient's rights under this Agreement terminate, Recipient agrees to cease use and distribution of the Program as soon as reasonably practicable. However, Recipient's obligations under this Agreement and any licenses granted by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in order to avoid inconsistency the Agreement is copyrighted and may only be modified in the following manner. The Agreement Steward reserves the right to publish new versions (including revisions) of this Agreement from time to time. No one other than the Agreement Steward has the right to modify this Agreement. The Eclipse Foundation is the initial Agreement Steward. The Eclipse Foundation may assign the responsibility to serve as the Agreement Steward to a suitable separate entity. Each new version of the Agreement will be given a distinguishing version number. The Program (including Contributions) may always be distributed subject to the version of the Agreement under which it was received. In addition, after a new version of the Agreement is published, Contributor may elect to distribute the Program (including its Contributions) under the new version. Except as expressly stated in Sections 2(a) and 2(b) above, Recipient receives no rights or licenses to the intellectual property of any Contributor under this Agreement, whether expressly, by implication, estoppel or otherwise. All rights in the Program not expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to this Agreement will bring a legal action under this Agreement more than one year after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.