Alessander Botti Benevides

General Report of Activities

Vitória - ES, Brazil
July 6, 2010
General Report of Activities

Research supervisor:
Giancarlo Guizzardi

Research supervisor:
João Paulo Andrade Almeida

ONTOMETRY AND CONCEPTUAL MODELING RESEARCH GROUP (NEMO)
COMPUTER SCIENCE DEPARTMENT
TECHNOLOGICAL CENTER
FEDERAL UNIVERSITY OF ESPIRITO SANTO

Vitória - ES, Brazil
July 6, 2010
Contents

List of Acronyms p. iv

List of Figures p. vi

List of Definitions p. vii

1 Introduction p. 1

1.1 Structure of the report p. 2

2 Background p. 3

2.1 Modal Logic p. 3

2.2 MDE Technologies p. 5

2.2.1 MDA p. 6

2.2.2 MOF p. 7

2.2.3 OCL p. 7

2.2.3.1 OCL Examples p. 9

2.2.4 Eclipse IDE p. 11

2.2.4.1 EMF p. 12

2.2.4.2 GMF p. 14

3 Unified Foundational Ontology and OntoUML Language p. 16

3.1 Classes and Generalization p. 17

3.2 Classifiers and Properties p. 29

3.3 Aggregation and Composition p. 39
## 4 A Tool for Building and Verifying OntoUML Models

4.1 Definition of the OntoUML Abstract Syntax in Ecore ................................... p. 49

4.2 Mapping the OntoUML Syntactical Constraints into OCL expressions ........... p. 49

4.3 Definition of the OntoUML Concrete Syntax by using GMF ........................... p. 51

4.4 Transforming the Ecore Metamodel and Additional OCL Constraints in a Graphical Editor ................................................................. p. 54

4.4.1 From the OntoUML profile to OntoUML Editor ..................................... p. 54

4.4.2 Licensing ................................................................................................. p. 55

## 5 Final Considerations

... p. 57

## Bibliography

... p. 59
# List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Arithmetic and Logic Unit</td>
</tr>
<tr>
<td>CASE</td>
<td>Computer-Aided Software Engineering</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Process Unit</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain-Specific Language</td>
</tr>
<tr>
<td>EER</td>
<td>Enhanced Entity-Relationship</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modeling Framework<a href="http://www.eclipse.org/modeling/emf">1</a></td>
</tr>
<tr>
<td>EPLv1</td>
<td>Eclipse Public License - v1.<a href="http://www.eclipse.org/org/documents/epl-v10.html">2</a></td>
</tr>
<tr>
<td>FOL</td>
<td>First Order Logic</td>
</tr>
<tr>
<td>FOSS</td>
<td>Free and Open Source Software</td>
</tr>
<tr>
<td>GEF</td>
<td>Graphical Editing Framework<a href="http://www.eclipse.org/gef">1</a></td>
</tr>
<tr>
<td>GMF</td>
<td>Graphical Modeling Framework<a href="http://www.eclipse.org/modeling/gmf">4</a></td>
</tr>
<tr>
<td>GPLv3</td>
<td>GNU General Public License Version 3<a href="http://www.gnu.org/licenses/gpl.html">5</a></td>
</tr>
<tr>
<td>GNU</td>
<td>GNU's Not Unix<a href="http://www.gnu.org">6</a></td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines<a href="http://www.ibm.com">7</a></td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>iff</td>
<td>if and only if</td>
</tr>
</tbody>
</table>

[1](http://www.eclipse.org/modeling/emf)
[2](http://www.eclipse.org/org/documents/epl-v10.html)
[3](http://www.eclipse.org/gef)
[4](http://www.eclipse.org/modeling/gmf)
[5](http://www.gnu.org/licenses/gpl.html)
[6](http://www.gnu.org)
[7](http://www.ibm.com)
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JET</td>
<td>Java Emitter Templates[^8]</td>
</tr>
<tr>
<td>MDA</td>
<td>Model-Driven Architecture[^7][^3]</td>
</tr>
<tr>
<td>MDE</td>
<td>Model-Driven Engineering</td>
</tr>
<tr>
<td>MDT</td>
<td>Model Development Tool[^10]</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta-Object Facility[^11][^2]</td>
</tr>
<tr>
<td>MVC</td>
<td>Model-View-Controller</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language[^4]</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group[^12]</td>
</tr>
<tr>
<td>OML</td>
<td>OPEN Modelling Language[^5]</td>
</tr>
<tr>
<td>OPEN</td>
<td>Object-oriented Process, Environment and Notation[^13]</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform-Independent Model[^14][^3]</td>
</tr>
<tr>
<td>PDM</td>
<td>Platform Definition Model[^15][^3]</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform-Specific Model[^16][^3]</td>
</tr>
<tr>
<td>UFO</td>
<td>Unified Foundational Ontology[^6]</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language[^17][^7][^8]</td>
</tr>
<tr>
<td>wff</td>
<td>well-formed formula</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange[^9]</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language[^18]</td>
</tr>
</tbody>
</table>

[^8]: http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html
[^9]: http://www.omg.org/mda
[^10]: http://www.eclipse.org/modeling/mdt
[^12]: http://www.omg.org
[^13]: http://www.open.org.au
[^14]: http://www.omg.org/mda
[^15]: http://www.omg.org/mda
[^16]: http://www.omg.org/mda
[^17]: http://www.uml.org
[^18]: http://www.w3.org/XML
List of Figures

1. EMF unifies Java, XML, and UML .................................................. p. 12
2. EMF framework ............................................................................. p. 13
3. EMF overview ............................................................................... p. 14
4. GMF overview ................................................................................ p. 15
5. UFO excerpt regarding Substantial Universals .............................. p. 18
6. Excerpt from the UML metamodel .............................................. p. 24
7. Revised fragment of the UML 2.0 metamodel ............................... p. 25
8. UFO excerpt regarding Relations, Moments, Quality Structures, etc. p. 29
9. Example of Quality Structures ......................................................... p. 31
10. Example of Characterization .......................................................... p. 32
11. Example of Relator Universal, Formal Relation, Material Relation, etc. p. 34
12. Excerpt of the UML metamodel featuring classifiers and Properties p. 35
13. Revised fragment of the UML 2.0 metamodel ......................... p. 36
14. UFO excerpt regarding meronymic relations ............................... p. 40
15. Revised fragment of the UML 2.0 metamodel ............................... p. 43
List of Definitions

1 Existential Dependence .................................................. p. 18
2 Rigidity ................................................................. p. 19
3 Anti-Rigidity ................................................................ p. 19
4 subtype ........................................................................... p. 20
5 covering ......................................................................... p. 21
6 disjoint ........................................................................... p. 21
7 partition .......................................................................... p. 21
8 Relational Dependence .................................................... p. 22
9 Phase Partition ............................................................... p. 23
10 Bearer of a Moment ......................................................... p. 30
11 Externally Dependent Mode ............................................. p. 32
12 Independence ................................................................. p. 32
13 Formal and Material Relations ......................................... p. 33
14 generic dependence ....................................................... p. 40
15 essential part ................................................................. p. 41
16 inseparable part ............................................................. p. 41
17 mandatory part .............................................................. p. 41
18 mandatory whole .......................................................... p. 41
19 exclusive part ............................................................... p. 43
20 general exclusive part-whole relation ............................. p. 43
1 Introduction

Mylopoulos[10] 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 (6) of a certain universe of discourse.

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 (6).

In pace with Degen et al.[11], we argue that “every domain-specific ontology must use as framework some upper-level ontology”. 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, in order 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 (12) [13].

An example of a general conceptual modeling and ontology representation language that has been designed following these principles is the version of Unified Modeling Language[7] [8] (UML) proposed in (6). This language (later termed OntoUML) has been constructed in a manner that its metamodel reflects the ontological distinctions prescribed by the Unified Foundational Ontology (6) (UFO). UFO is a foundational ontology designed specially for

[http://www.uml.org]
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 OntoUML’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.

Although the OntoUML language has been able to provide mechanisms for addressing a number of classical conceptual modeling problems (14), and the language has been successfully employed in application domains (15), (16), there was still no tool support for building and verifying conceptual models and domain ontologies constructed using OntoUML. The main contribution of the work reported here will be to present a model-based OntoUML Graphical Editor with support for automatic model checking in face of ontological constraints.

1.1 Structure of the report

Besides this introductory chapter, this report is organized in five additional chapters. Chapter 2 contains a theoretical background about the system of modal logics employed in this report and a number of Model-Driven Engineering (MDE)-related technologies that are relevant for this work. Chapter 3 contains an introduction on UFO and OntoUML. Chapter 4 is about building a tool for model building and verification. Finally, in chapter 5 we pose our final considerations.
2 Background

In this chapter we present the theoretical background for the research reported here. In section 2.1 we briefly comment on the system of modal logics employed in this report. Section 2.2 presents a number of MDE-related technologies that are relevant for this research, namely, Meta-Object Facility (MOF) and Object Constraint Language (OCL) (in this order). Moreover, the same subsection also presents the Eclipse Integrated Development Environment (IDE) and a number of its plug-ins (Eclipse Modeling Framework (EMF), Graphical Modeling Framework (GMF) and Model Development Tools (MDT)), which are important to achieve the purposes of this work.

2.1 Modal Logic

In this report, we write our logical expressions in First Order Logic (FOL) or in a Quantified Modal Logics with Identity. As FOL is well known, we will only describe precise semantics for 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 (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 (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. (p. 20)
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 (19). 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 possible 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 (20). 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 \( \varepsilon \) can then be explicitly defined such that \( \varepsilon(x) \triangleq \exists y (y = x) \).

For the modal propositions created in the present report, we make use of a language \( L \) of quantified modal logics with identity. The alphabet of \( L \) contains the traditional operators of \( \land \) (conjunction), \( \lor \) (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) \( \Diamond A \triangleq \neg \Box \neg A \); (2) \( \Box A \triangleq \neg \Diamond \neg A \) and (3) \( \exists! x(A) \triangleq \exists y (\forall x (A \leftrightarrow (x = y))) \). Additionally, we add that the models assumed here are the so-called normal models (21), 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 $\delta$ that assigns values to the non-logical constants of the language and a model structure $M$. In this language $M$ has a structure $\langle K, D \rangle$ where $K$ is a Kripke structure, and $D$ 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 domain 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 $\Box A$ is true in world $w$ if and only if (iff) $A$ is true in every world $w'$ accessible from $w$. Likewise, the semantic value of formula $\Diamond A$ is true in world $w$ iff $A$ is true in at least one world $w'$ accessible from $w$.

Finally, in chapter 3 following the original formal characterization of UFO and OntoUML language (6), we assume all worlds to be equally accessible and, as a result, we have the language of a possibilist quantified modal logic QS5.

### 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 OS/360 and Unix, shielded developers from complexities of programming directly to hardware (22, 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 (22, 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.

Now, 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) (23, 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 (24, 25).

Now, the field of MDE aims at consider models as first class entities, providing standards for metamodeling, such as the Model-Driven Architecture (3) (MDA) (subsection 2.2.1) and MOF and OCL languages (see subsections 2.2.2 and 2.2.3 respectively), as well as tools, such as the Eclipse (subsection 2.2.4) and some of its plug-ins, like EMF (subsubsection 2.2.4.1), GMF (subsubsection 2.2.4.2) and MDT (subsection 2.2.4).

2.2.1 MDA

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

The MDA is related to a number of standards, such as UML, OCL, MOF and XML Metadata Interchange (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, that we make extensive use in the present research. Eclipse is briefly presented in subsection 2.2.4.

---

5 http://www.omg.org/mda
6 http://www.omg.org
7 http://www.omg.org
8 http://www.omg.org/mda
9 http://www.omg.org/mda
10 http://www.omg.org/mda
2.2 MDE Technologies

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 is available at [2] (26, 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 (27).

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 (2) (EMOF) language (see section 2.2.4.1).

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 is 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 (4) (OCL), have been developed (4, p. 5).

OCL has been developed as a business modeling language within the International Business Machine (IBM) Insurance division, and has its roots in the Syntropy method (28) (4, p. 5), being based on a set theory and predicate logics and having a formal mathematical semantics (29).

OCL is a pure specification language. Therefore, an OCL expression is guaranteed to be without side effects (4, 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.

---

11 http://www.ibm.com
12 “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)” (4, p. 5).
2.2 MDE Technologies

(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 (4, p. 5). This makes 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 (29).

In UML 2.0, OCL can be utilized not only to specify constraints, but also to specify some other expressions in the elements of the diagram. 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 (29).

As stated by 4, pp. 5-6, OCL 2.0 can be used for a number of different purposes:

• 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.

Additionally, 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 (7) and MOF 2.0 Core (2), because the full OCL specification can only be used with the UML language (4, pp. 1,171). The Essential OCL language is defined in 4, pp. 171-175. As we use the Ecore language (for Ecore, see section 2.2.4.1) to build the OntoUML metamodel, then all OCL expressions that we construct in this report are Essential OCL expressions.
2.2.3.1 OCL Examples

Now, we will get our examples of the OCL syntax from the tutorial [30]. 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: \(*\), \(+\), \(-\), \(/\), \(\text{div}()\), \(\text{abs}()\), \(\text{mod}()\), \(\text{max}()\), \(\text{min}()\), \(\text{sum}()\), \(\text{sin}()\) and \(\text{cos}()\).
- Real: \(*\), \(+\), \(-\), \(/\), \(\text{floor}()\), \(\text{sum}()\), \(\text{sin}()\) and \(\text{cos}()\).
- Boolean: \(\text{and}\), \(\text{or}\), \(\text{xor}\), \(\text{not}\), \(\text{implies}\) and \(\text{if-then-else}\).
- String: \(\text{concat}()\), \(\text{size}()\), \(\text{substring}()\), \(\text{toInteger}()\) and \(\text{toReal}()\).

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

The iterators over collections:
• The selection of a sub-collection: select().

• When specifying a collection which 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 ).
2.2 MDE Technologies

- Concatenating Sequences: Sequence\{1, 2, 3\} \rightarrow \text{union}(Sequence\{4, 5, 6\}).

\text{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 \text{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 \text{oclIsKindOf()} checks if a given instance is an instance of a certain type or of one of its subtypes.
- The operation \text{allInstances()} returns you all instances of a given Type.
- The operation \text{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 \text{n} are often represented with collections, which may be empty and not void).

Examples on \text{OCL} class operations:

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

Finally, \text{OCL} comments start with two consecutive hyphens (“--”) and end at the end of the line.

2.2.4 Eclipse IDE

We use the Eclipse IDE mainly because it provides, in form of plug-ins, a set of functionalities that are useful 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 (31):

“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.” (31)
In the Eclipse IDE, some functionalities are provided via independent frameworks, implemented as plug-ins. In this work, we have used:

- Eclipse Modeling Framework\(^{13}\) (EMF);
- Graphical Modeling Framework\(^{14}\) (GMF);
- Model Development Tools\(^{15}\) (MDT);

### 2.2.4.1 EMF

In order to create the OntoUML metamodels, we use 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. 1. 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 (26, p. 14).

![Figure 1: EMF unifies Java, XML and UML (26, p. 14).](image)

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). However, EMF can transparently read and write serializations of EMOF\(^{32}\).

In the words of 26, p. 39:

> MOF and Ecore have many similarities in their ability to specify classes and their structural and behavioral features, inheritance, packages, and reflection.

\(^{13}\)http://www.eclipse.org/modeling/emf
\(^{14}\)http://www.eclipse.org/modeling/gmf
\(^{15}\)http://www.eclipse.org/modeling/mdt
\(^{16}\)http://www.w3.org/XML
They differ in the area of life cycle, data type structures, package relationships, and complex aspects of associations.\textit{(26) p. 39}"

Now, we can update Fig. 1 in the Fig. 2.

![Diagram](http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html)

Figure 2: EMF framework \textit{(26) 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.

Additionally, as only Essential OCL can be used with EMOF (meta)models (see section \textit{2.2.3}), then all OCL expressions that we build in this work are Essential OCL expressions.

Fig. 3 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 \textit{(1) p. 47}. If the metamodel contains OCL expressions, then, as is described in \textit{34}, we have to set some variable names in this Genmodel file, regarding the use of some Java Emitter Templates\textsuperscript{17} (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.

Following \textit{35} from the Genmodel, we perform two automatic transformations in order to get the Java code for the Ecore file and an EMF Edit \textit{(36)} framework, which provides generic reusable classes, which we use to build the graphical editor.

\textsuperscript{17}http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html
2.2 MDE Technologies

2.2.4.2 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 (GEF) framework, which is a plug-in that allows the construction of graphical editors from Ecore metamodels, but of difficult utilization. Fig. 3 shows an overview of the process of creating a graphical editor by using the GMF plug-in.

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.

18 http://www.eclipse.org/gef
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 the syntactical constraints of the language, which can be verified in live or in batch mode; these verification modes are explained in sections ?? and ??, respectively.

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 of the Java code that implements the editor, by the GMF framework.
3 Unified Foundational Ontology and OntoUML Language

Guizzardi, in [6], 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” (6). In [6] the author also does exactly that by proposing an ontologically well-founded version 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] (6, 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 (14, p. 123) (see section ??) 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 also part of y, see section ??).

In order to formally assess the quality of [UML 2.0] (6, 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

---

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.
3.1 Classes and Generalization

construct overload, which leads to ambiguity in the produced models. If there are constructs in 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 (6).

In the following, we show the results obtained in [6] 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 [6] 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 [6] pp. 311-352.

In the assessment of UML 2.0, it was used only a fragment of the UFO ontology regarding endurants (6, p. 211), i.e., objects. As is done in chapter 8 of [6] 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 modelling language whose metamodel is designed to be isomorphic to the UFO ontology, the leaf ontological distinctions in the metamodel of UFO appear as modelling primitives in the language.

### 3.1 Classes and Generalization

The Fig. 5 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 the color of a table) or Relations.
3.1 Classes and Generalization

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 which can be briefly defined as follows:

**Definition 1 (Existential Dependence):** Let the predicate $\varepsilon$ denote existence.\footnote{In an actualist system, the existence operator $\varepsilon$ can be explicitly defined such that $\varepsilon(x) \triangleq \exists y(y = x)$.} We have that an individual $x$ is existentially dependent on another individual $y$ as a matter of necessity, $y$ must exist whenever $x$ exists, or formally: $ed(x, y) \triangleq \Box(\varepsilon(x) \rightarrow \varepsilon(y))$. (38, 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 for 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 $\forall x, y(\text{Substance}(x) \land \text{Substance}(y) \land disjoint(x, y) \rightarrow \neg ed(x, y) \land \neg ed(y, x))$.

---

Figure 5: UFO excerpt regarding Substantial Universals (6, p. 315).
3.1 Classes and Generalization

Also, from [6] 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 (6, p. 213) (see section 3.2 for more details) and to which it is existentially dependent. A moment can inhere on one single individual (e.g., the color of a table, 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 provides a principle of individuation and identity to its instances (its particulars) (6, 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(T) \Leftrightarrow \Box(\forall x(T(x) \rightarrow \Box(T(x)))). \quad (38) \text{ 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(T) \Leftrightarrow \Box(\forall x(T(x) \rightarrow \Diamond(\neg T(x)))). \quad (38) \text{ 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 (6, 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) (6, p. 108)(e.g., Woman, which inherits its identity principle from Person).

Regarding generalizations, both [7] p. 64 and [8] 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.” (7, p. 64), (8, p. 72)

Therefore, based on 39, 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 $\omega$ in which $x$ is an instance of $C_1$: $\text{subtype}(C_1, C_2) \triangleq \Box(\forall x(C_1(x) \rightarrow C_2(x)))$. ■

Regarding the GeneralizationSet meta-attributes $\text{isCovering}$ and $\text{isDisjoint}$, (8, p. 75) defines them in the following way:

- “$\text{isCovering}$ : Boolean
  
  Indicates (via the associated Generalizations) whether or not the set of specific Classifiers are covering for a particular general classifier. When $\text{isCovering}$ 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 $\text{isCovering}$ 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 either 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 $\text{false}$. ■

- “$\text{isDisjoint}$ : Boolean
  
  Indicates whether or not the set of specific Classifiers in a Generalization relationship have instance in common. If $\text{isDisjoint}$ is true, the specific Classifiers for a particular GeneralizationSet have no members in common; that is, their intersection is empty. If $\text{isDisjoint}$ 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.” (S p. 75)

Also, when a GeneralizationSet has both isCovering and isDisjoint meta-attributes being true, then it is known as a “partition” (S p. 78).

Therefore, we define covering, disjoint and partition GeneralizationSets that contain generalizations between non-relational Classifiers, such as a Classes or a 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 $C_1, \ldots, C_n$ ($n \in \mathbb{N}^*$) that are the subtypes of $C_{ST}$ that are referred by the Generalizations referred by GS. When 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 $\{C_1, \ldots, C_n\}$: $\text{CoveringGeneralizationSet}(C_{ST}, C_1, \ldots, C_n) \triangleq ((\bigwedge_{1 \leq i \leq n} C_i(x) \rightarrow C_{ST}(x))) \land (C_{ST}(x) \rightarrow (\bigvee_{1 \leq j \leq n} C_j(x)))$.

- **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 $C_1, \ldots, C_n$ ($n \in \mathbb{N}^*$) that are the subtypes of $C_{ST}$ that are referred by the Generalizations referred by GS. When 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 $C_j$ ($j \in \{1, \ldots, n\} - \{i\}$): $\text{DisjointGeneralizationSet}(C_{ST}, C_1, \ldots, C_n) \triangleq (\bigwedge_{1 \leq i \leq n} (C_i(x) \rightarrow C_{ST}(x)) \land (\bigwedge_{1 \leq j \leq n, j \neq i} \neg C_j(x)))$.

- **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 $C_1, \ldots, C_n$ ($n \in \mathbb{N}^*$) that are the subtypes of $C_{ST}$ that are referred by the Generalizations referred by GS. When both 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 $\{C_1, \ldots, C_n\}$: $\text{PartitionGeneralizationSet}(C_{ST}, C_1, \ldots, C_n) \triangleq ((\bigwedge_{1 \leq i \leq n} C_i(x) \rightarrow C_{ST}(x)) \land (C_{ST}(x) \rightarrow (\bigoplus_{1 \leq j \leq n} C_j(x))))$, 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 (e.g., Person). Quantities are maximally self-connected objects (e.g., Sand). Collectives are collections of entities connected by an unifying relationship (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 (e.g., in Fig. ??, a Child is a Living Person whose age is within a certain range; likewise, a Living Person 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 (p. 294), for an explanation on the mediation relation and the metaclass Relator, see section 3.2), as the participation in an event or relationship (e.g., in Fig. ??, the Role Student, played by a Living Person 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 for every instance x of T, there is an instance y of P such that x and y are related via R: R(T, P, R) ≜ □∀x(T(x) → ∃y(P(y) ∧ R(x,y)))). ■ (p. 10)

Also, as discussed in p. 103, phases are always defined in a partition set:

“Definition 4.1 (Extension functions): Let $W$ be a non-empty set of possible worlds and let $w \in W$ be a specific world. The extension function $ext_w(G)$ 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. $\forall w \in W \:\:\: ext_w(G) = \bigcup_{w \in W} ext_w(G)$ (pp. 100, 101, Definition 4.1))

2. $\forall w \in W \:\:\: ext_w(S) = \bigcup_{P_i \in \langle P_1, \ldots, P_n \rangle} ext_w(P_i)$

and for all $P_i, P_j \in \langle P_1, \ldots, P_n \rangle$ (with $i \neq j$) we have that

3. $ext_w(P_i) \cap ext_w(P_j) = \emptyset$ (p. 103)
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. ??, the universals Child, Teenager and Adult define a phase partition for the Phase LivingPerson. As consequence, we have that in 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 [6] 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 \( P_i \), i.e., for any \( P_i \in P_1 \ldots P_n \), which restricts \( S \), and for any instance \( x \) such that \( x \in \text{ext}_w(S) \), there is a world \( w^f \in W \) such that \( x \in \text{ext}_{w^f}(P_i) \). This is equivalent of stating that for any \( P_i \in P_1 \ldots P_n \), the following holds

11. \( \text{ext}(S) = \text{ext}(P_i) \)” ([6] 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 \( SC \) which is partitioned in Phase subclasses \( \langle P_1, \ldots, P_n \rangle \), then, for every \( P_i \) such that \( i \in \{1, \ldots, n\} \) there must exist at least one world \( w^j \) in which \( x \) instantiates \( P_i \): \( \text{PhasePartition}(SC, P_1, \ldots, P_n) \triangleq (\bigwedge_{1 \leq i \leq n} (P_i(x) \rightarrow SC(x) \land (\bigwedge_{1 \leq j \leq n} \bigwedge_{j \neq i} P_j(x)))) \land (SC(x) \rightarrow (\bigoplus_{1 \leq k \leq n} P_k(x)))) \). ■

In summary, in the model depicted in Fig. ??, 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, [14] 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 ([6] 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 ([6] p. 112) (e.g., RelationalEntity as a generalization of Person and
3.1 Classes and Generalization

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 (6, 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 (6, 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 6, pp. 28-36 and using the excerpt of UFO shown in Fig. 5 (6, p. 314) revise the excerpt of the UML 2.0 metamodel shown in Fig. 6 (6, p. 312) creating the excerpt of the OntoUML metamodel pictured by Fig. 7.

Figure 6: Excerpt from the UML metamodel featuring the metaclasses Classifier, Class, Generalization and GeneralizationSet (6, p. 312).

OntoUML profile regarding the categories depicted in Fig. 7

Metaclass: Substance Sortal
Figure 7: Revised fragment of the UML 2.0 metamodel according to the ontological categories of Fig. 5 (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
3.1 Classes and Generalization

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»
3.1 Classes and Generalization

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 Lepidopterum.

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.
3.1 Classes and Generalization

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.
3.2 Classifiers and Properties

The Fig. 8 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. 5) 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 8: UFO excerpt regarding Relations, Moments, Quality Structures and related categories (6, 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 Data_Type construct” (6, p. 326). This concept is specialized into Quality Dimensions and Quality Domains.

Constraints:

1. A «mixin» cannot have a «roleMixin» as a supertype.
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. 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 \( i(x,y) \) and implies existential dependence (see Definition 1) from the instances of Moment Universals to other individuals, named their bearers (6, p. 213). **Definition 10 (Bearer of a Moment):** The bearer of a moment \( x \) is the unique individual \( y \) such that \( x \) inheres in \( y \). Formally, \( \beta(x) \equiv \exists y \ i(x,y) \). Existential dependence can be used to differentiate Intrinsic Moment Universals and Relator Universals: instances of Intrinsic Moment Universals are dependent of one single individual; instances of Relator Universals depend on a plurality of individuals (6, 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 (6, p. 224) by means of Attribute Functions. A Mode Universal is a Intrinsic Moment Universal that is not directly related to Quality Structures (6, 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.” (6, p. 325)

In order to exemplify Quality Dimensions, Quality Domains, Quality Universals and Attribute Functions, let us create an OntoUML model, pictured in Fig. 9, 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...
3.2 Classifiers and Properties

Associated with Age, and in the context of a given measurement system, there is a Quality Dimension \( \text{ageValue} \) which is a linear structure isomorphic to the natural numbers \((\text{age} \in \mathbb{N})\) obeying the same ordering structure. Thus, we represented the Quality Dimension \( \text{ageValue} \) as the Simple Datatype \( \text{NaturalNumber} \) shown in Fig. 9. In this case, we can define an Attribute Function \( \text{age(Year)} \), which maps each instance of Person (and in particular \( x \)) onto a point in the Quality Dimension \( \text{ageValue} \) (in OntoUML, we represented the Attribute Function \( \text{age(Year)} \) as a Datatype Relationship with “age” as its navigable end name, from the Kind Person to the Simple Datatype \( \text{NaturalNumber} \)). 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 \( \text{birthdayValue} \) which contains three Quality Dimensions, namely, \( \text{dayValue} \), \( \text{monthValue} \) and \( \text{yearValue} \) 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\(^7\)). 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 \( \text{birthdayValue} \) as the Structural Datatype \( \text{Birthday} \), which has three Datatype Relationships (with “day”, “month” and “year” as the navigable end names) to the Simple Datatype \( \text{NaturalNumber} \). In this case, we can define an Attribute Function \( \text{birthday} \), which maps each instance of Person (and in particular \( x \)) onto a vector in the corresponding 3-dimensional Quality Domain \( \text{birthdayValue} \) (in OntoUML, we represented the Attribute Function \( \text{birthday} \) as a Datatype Relationship with “birthday” as its navigable end name, from the Kind Person to the Structural Datatype \( \text{Birthday} \)).

Figure 9: 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 \( y::\text{IMU} \) such that

\(^7\)After choosing a calendar system, those constraints could be written as OCL expressions in the model shown in Fig. 9.
3.2 Classifiers and Properties  

For example, the Mode Universal Mental State characterizes the Kind 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. [10].

![Figure 10: Example of Characterization.](image)

Relator Universals are aggregations of all \textit{qua individuals} that share the same foundation (6, p. 240). A \textit{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] (6, pp. 218,238) in the following way:

**Definition 11 (Externally Dependent Mode):** A mode \( x \) is externally dependent it is existentially dependent of an individual which is independent of its bearer (see Definition [10]). Formally, \( \text{ExtDepMode}(x) \equiv \text{Mode}(x) \land \exists y \text{ indep}(y, \beta(x)) \land \text{ed}(x, y) \) ■ (6, p. 238)

**Definition 12 (Independence):** \( \text{indep}(x, y) \equiv \neg \text{ed}(x, y) \land \neg \text{ed}(y, x) \) ■ (6, p. 218)

Intuitively, a \textit{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) (6, p. 239).

A Relator Universal RU mediate another Substantial Universal SU by means of a mediation relation, symbolized by \( \text{mediation}(SU, RU) \). If \( x, y \) and \( z \) are three distinct individuals such that: (i) \( x::RU \); (ii) \( y \) is a \textit{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 \( m::\text{mediation} \) (6, pp. 240-241). Also, as said above, if we have \( i(y, z) \) then \( \text{ed}(y, z) \), and as \( y \) is a \textit{part of} \( x \), then \( \text{ed}(x, z) \) (considering that existential dependence is transitive from parts to wholes[^9]). Therefore, \( m(x, z) \) implies \( \text{ed}(x, z) \). Also, every instance of a Relator 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”(6, p. 236) (e.g., the Formal Relation olderThan, derived from the property age of the type Person;[^8]

[^8]: \( \text{ed}(x,y) \) is defined in Definition [1]
[^9]: Besides it is not stated in [6] we consider feasible to suppose that \( \forall x,y,z ((\text{ed}(x,y) \land \text{part}_o f(x,z)) \rightarrow \text{ed}(z,y)) \)
3.2 Classifiers and Properties

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 (6, p. 236).

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 \(U_1, \ldots, U_n\) if

\[
\forall a_1, \ldots, a_n R(a_1, \ldots, a_n) \iff \bigwedge_{i \leq n} U_i(a_i) \land \varphi(a_1, \ldots, a_n)
\]

A Relation is a Material Relation if there is a Relator Universal \(U_R\) such that the condition \(\varphi\) is obtained from \(U_R\) as follows:

\[
\varphi(a_1, \ldots, a_n) \iff \exists k U_R(k) \bigwedge_{i \leq n} m(k, a_i)
\]

In this case, we say that the relation R is derived from the relator universal \(U_R\), or symbolically, derivation\((R, U_R)\). Otherwise, if such a Relator Universal \(U_R\) 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 \(R\) if 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 \(\bullet \rightarrow \bullet \rightarrow \ldots \rightarrow \bullet\), 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. 11, 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.
3.2 Classifiers and Properties

Figure 11: Example of Relator Universal, Formal Relation, Material Relation, mediation and derivation.

Again, by applying the framework for assessment of modeling languages proposed in [6], pp. 28-36 and using the excerpt of UFO shown in Fig. 8 [6], p. 323 revise the excerpt of the UML 2.0 metamodel shown in Fig. 12, creating the excerpt of the OntoUML metamodel pictured by Fig. 13.

OntoUML profile regarding the categories depicted in Fig. 13

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.
3.2 Classifiers and Properties

Figure 12: Excerpt of the UML metamodel featuring classifiers and Properties (6 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.
3.2 Classifiers and Properties

Figure 13: Revised fragment of the UML 2.0 metamodel according to the ontological categories of Fig. 8 [6] p. 334).

Constraints:

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 ≥ 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 ≥ 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 ≥ 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.
3.2 Classifiers and Properties

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 previously shown in this section. 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 this section. 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. [14] 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. [8]) 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 (6, 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 (5) (OML), UML Enhanced Entity-Relationship (EER)). An important aspect to be addressed by any
3.3 Aggregation and Composition

3.3.1 Conceptual Theory of Parthood

Figure 14: UFO excerpt regarding meronymic relations (6, 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 (38), 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 $T$ whenever $y$ exists it is necessary that an instance of $T$ exists. This can be formally characterized by the following formula schema: $GD(y, T) \equiv \Box(\epsilon(y) \rightarrow \exists T, x(\epsilon(x)))$ [38, 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.

---

10This definition is formalized in a language of modal logics defined in [6] and in which all quantification is restricted by Sortals (6, pp. 121-122), so $\exists T, x(A)$ 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 \( y \):  
\[
EP(x, y) \equiv ed(y, x) \land (x \leq y).
\]

This is equivalent to stating that  
\[
EP(x, y) \equiv \Box((\epsilon(y) \rightarrow \epsilon(x)) \land (x \leq y)).
\]

We adopt here the mereological continuism defended by 41, which states that the part-whole relation should only be considered to hold among existents, i.e.,  
\[
\forall x, y((x \leq y) \rightarrow (\epsilon(x) \land \epsilon(y))).
\]

As a consequence, we can have this definition in its final simplification:  
\[
EP(x, y) \equiv \Box(\epsilon(y) \rightarrow (x \leq y)).
\]

**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 \( y \):  
\[
IP(x, y) \equiv \Box(\epsilon(x) \rightarrow (x \leq y)).
\]

**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 \( T \):  
\[
MP(T, y) \equiv \Box(\epsilon(y) \rightarrow \exists T, x(x < y)).
\]

**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 \( T \):  
\[
MW(T, x) \equiv \Box(\epsilon(x) \rightarrow \exists T, y(x < y)).
\]

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), immutable regarding the part or immutable regarding the whole;

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. 5 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 (6, 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.” (6, p. 350). ComponentOf relations are non-reflexive anti-symmetric non-transitive parthood relations, which obeys the weak supplementation principle (6, 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.” (6, p. 350). SubQuantityOf relations are essential (see Definition 15) non-shareable (see Definition 20) non-reflexive anti-symmetric transitive parthood relations, which obeys the strong supplementation principle and the extensionality principle (6, 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.” (6, 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.” (6, p. 351). SubCollectionOf relations are non-reflexive anti-symmetric transitive parthood relations, which obeys the weak supplementation principle (6, 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.” (6, p. 352). MemberOf relations are non-reflexive anti-symmetric intransitive parthood relations, which obeys the weak supplementation principle (6, p. 352).

Moreover, all parthood relations can be characterized regarding their shareability. (8, p. 39) is intentionally vague when stating that:

“Precise semantics of shared aggregation varies by application area and modeler. The order and way in which part instances are created is not defined.”. (8, p. 39)

From (6, 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

---

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 $y$ which does not overlap with $x$ (6, 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 (6, p. 147).
3.3 Aggregation and Composition

exclusive (proper) part of another individual $y$ of type B (symbolized as $<_X (x, A, y, B)$) if $y$ is the only B that has $x$ as part.  

$<_X (x, A, y, B) \equiv (x :: A) \land (y :: B) \land (x < y) \land (\forall z :: B ((x < z) \rightarrow (y = z)))$  

(6, 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 $A <_{GX} B$) if every instance $x$ of A has an exclusive part of type B.  

$A <_{GX} B \equiv \forall x (x :: A) \rightarrow \exists y (y :: B) \land (x < y) \land _X (x, A, y, B)$  

or simply,

$A <_{GX} B \equiv \forall x (x :: A) \rightarrow \exists ! y (y :: B) \land (x < y)$  

(6, p. 162)

Finally, by applying the framework for assessment of modeling languages proposed in (6) pp. 28-36 and using the excerpt of UFO shown in Fig. 14 (6) p. 341 revise the excerpt of the UML 2.0 metamodel shown in Fig. 12 creating the excerpt of the OntoUML metamodel pictured by Fig. 15.

Figure 15: Revised fragment of the UML 2.0 metamodel according to the ontological categories of Fig. 14 (6) p. 348).

OntoUML profile regarding the categories depicted in Fig. 15

Metaclass: Meronymic
3.3 Aggregation and Composition

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 \( \text{lower}_{C_i} \) 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 \textit{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 \textit{isEssential} must be false, whereas the meta-attribute \textit{isImmutable} may be true. However, if \textit{isEssential} is true (in case of a rigid classifier with essential parts) then \textit{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 \textit{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 \textit{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 \( \text{isShareable} = \text{true} \) 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 \textbf{ALU} is part of a \textbf{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, an 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.
In order to accomplish our goals of building an editor capable of automatic syntax checking and automatic model filling, we employ Model-Driven Architecture (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 (OMG) MOF metamodeling architecture, the languages Ecore, OCL, the Eclipse platform, and some of its plug-ins, mainly EMF, MDT, and GMF.

In general, we will perform the following tasks in the construction of the editor:

1. Implement the OntoUML abstract syntax (metamodel) 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 the automatic calculation of Material Associations and Derivations\(^1\) cardinals (p. 331, Fig. 8-10) as OCL expressions;


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

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

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

\(^1\)For these OntoUML concepts, see chapter 3
4.1 Definition of the OntoUML Abstract Syntax in Ecore

- A set of OCL expressions formalizing the automatic calculation of Material Associations and Derivations cardinalities;
- A set of OCL expressions formalizing the OntoUML syntactical constraints;
- A GMF definition of the OntoUML concrete syntax;

In the following sections, we explain how to accomplish each one of the tasks described above: section 4.1 shows how we will build an unified Ecore metamodel from the three fragments of the OntoUML metamodel pictured in Figs. 7, 13 and 15 in order to define the abstract syntax of the OntoUML language by using the Eclipse Modeling Framework (EMF) plug-in. Section 4.2 shows how we will address the implementation of the OntoUML syntactical constraints (or well-formedness rules) taken from page 317–320, 334–338, 348–352 as OCL expressions. The section 4.3 shows a possible definition of the OntoUML concrete syntax (as specified in page 317–320, 334–338, 348–352) by using the Graphical Modeling Framework (GMF) plug-in. In the section 4.4 we show the set of transformations from the Ecore metamodel (embedded with OCL constraints), which will lead to the creation of the Java code implementing the OntoUML Graphical Editor.

4.1 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 Model-View-Controller (MVC) architecture, we have to define the OntoUML’s metamodel in a language named Ecore.

So, we will use the EMF plug-in to create an Ecore metamodel for OntoUML by unifying the three fragments of the OntoUML metamodel pictured in Figs. 7, 13 and 15 in a unique metamodel. Also, we have to create OCL expressions in order to indicate how the derived meta-attributes and meta-relations of the OntoUML metamodel get their values.

4.2 Mapping the OntoUML Syntactical Constraints into OCL expressions

All the syntactical constraints of the OntoUML language will be defined as OCL expressions in order to make use of the verification framework provided by the GMF Eclipse plug-in.

---

2 http://www.eclipse.org/modeling/emf
3 http://www.eclipse.org/modeling/gmf
In general, there are two types of syntactical constraints regarding the way in which they are verified:

- constraints that prevent 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 prevent 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 than 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.

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.

- 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.

- Constraints on the unchangeability of some values of determined meta-attributes of specific Classifiers. As the values of these meta-attributes shall always be initialized with the correct value, their unchangeability can be verified in *live* mode. An example is the unchangeability of the meta-attribute isAbstract=true for the MixinClasses.

- 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.
• Constraints on the cardinalities of specific types of OntoUML relationships that shall be 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 shall be automatically calculated in the moment of the creation of the relations, we have to prohibit the user from entering inconsistent values.

• 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 \(*\).

The OntoUML syntactical constraints that are \textit{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, \textit{e.g.}, every non-abstract (isAbstract=false) non-SubstanceSortal must have a SubstanceSortal as its supertype.

• Constraints about the conditions that are specified by meta-attributes, like the isExtensional meta-attribute of the metaclass Collective, which specifies that all of its parts must be essential (isEssential=true).

• Constraints about the existence of covering and disjoint GeneralizationSets relating the Phases of a Sortal, because the Phases must partition a Sortal.

• Constraints about the relational dependence, \textit{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.

• 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.

4.3 Definition of the OntoUML Concrete Syntax by using GMF

In this chapter, we discuss how the OntoUML concrete syntax (defined in the OntoUML profile \(\text{p. 317–320, 334–338, 348–352}\)) can be mapped in the GMF technologies for the creation of
4.3 Definition of the OntoUML Concrete Syntax by using GMF

the graphical editor.

As discussed in the subsubsection 2.2.4.2, 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 connect 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 contain 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 will 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 can model the OntoUML concrete syntax in the following way:

1. The metaclasses Mode, Relator, Category, Mixin, RoleMixin, Collective, Kind, Quantity, Phase, Role, simpleDatatype and structuralDatatype can be 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} for its meta-attribute isExtensional;
2. The metaclass GeneralizationSet can be 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 can be 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

---

4In general, in a label, the text that is related to a boolean meta-attribute is only visible when the meta-attribute is true.
4.3 Definition of the OntoUML Concrete Syntax by using GMF

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

4. The metaclass Mediation can be 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 can be 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 can be 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 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);

7. The metaclass DatatypeRelationship can be 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 can be represented as a Connection with a solid line shape decorated in the source with a closed arrow;

9. The metaclass Property can be 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 datatypeRelationships.

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 can be modeled by the standard visualization of stereotypes and tagged

---

5This label has a distinct behaviour: when isEssential is true then 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 isEssential is false and isImmutablePart is 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.
4.4 Transforming the Ecore Metamodel and Additional OCL Constraints in a Graphical Editor

In this section we describe how we will implement the editor by using some Eclipse plug-ins. Subsection 4.4.1 shows how OntoUML Editor will be implemented; subsection 4.4.2 deals with licensing the source code.

4.4.1 From the OntoUML profile to OntoUML Editor

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

Then, we can automatically transform this Ecore metamodel into a Genmodel file. In this Genmodel, we have to 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 34. From the Genmodel we have to perform two automatic transformations (following 35) and get the Java code for the Ecore metamodel and an EMF.Edit framework, which provides generic reusable classes, which we will use for building our graphical editor.

Thereon, we have to create a GMF project and perform more transformations on the Ecore metamodel. From the Ecore metamodel, we will create a GMFGraph file, which will describe 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 will indicate which metaclasses or meta-attributes will be the nodes, the connections or the labels. We will create the OntoUML stereotypes as labels of the metaclasses that are defined as nodes.

From the Ecore metamodel, we also have to create a GMFTool file, in which, again, we will select which metaclasses or meta-attributes will be the nodes, the connections or the labels. In this file, we will manually create the names of the tools that will be in the tool palette of our editor and which will be utilized in order to instantiate the OntoUML constructs (metaclasses or
4.4 Transforming the Ecore Metamodel and Additional OCL Constraints in a Graphical Editor

meta-attributes) in nodes or connections.

In order to create a mapping between the Ecore metamodel, the GMFGraph file and the GMFTool file, we have to create a GMFMap file. In this file, we will map the elements from the Ecore metamodel to their visualization specification (which will be defined in the GMFGraph file) and their creation tools specification (which is defined in the GMFTool file). In order to create this file, we have to 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 have to select which metaclasses or meta-attributes will be the nodes, the connections or the labels. In this GMFMap file, we will also put the OCL constraints, which can be verified in live or in batch mode (see subsection ?? for live and batch constraints). In order to create live constraints, we will put them inside “Audit Rules” that have their “Use In Live Mode” properties set to true. Otherwise, in order to create batch constraints, we will put them inside “Audit Rules” that have their “Use In Live Mode” properties set to the default false value.

Once the changes in the GMFMap file are finished, we will 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 have to set some properties in order to enable the verification of OCL constraints in OntoUML models: we have to 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 will generate the Java code that implements the OntoUML Editor.

4.4.2 Licensing

This graphical editor, named OntoUML Editor, will be a Free and Open Source Software (FOSS) and its source code will be licensed under GNU General Public License Version 3 GPLv3 (43) and, occasionally, Eclipse Public License - v1.0 EPLv1 (44). 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 (45).

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 (46)), 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.
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, these 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.

In this report, we present the project of an Eclipse-based graphical editor which aims at fulfilling the gap of tool support for one particular theoretically well-founded representation language, namely, OntoUML. Underlying this editor there will be an implementation of the OntoUML metamodel proposed in [6] 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.

Finally, we believe that the primary contribution of the present work will be 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;
Bibliography


