UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
DEPARTAMENTO DE INFORMÁTICA
MESTRADO EM INFORMÁTICA

CLAUDENIR MORAIS FONSECA

ML2: AN EXPRESSIVE MULTI-LEVEL CONCEPTUAL MODELING LANGUAGE

VITÓRIA-ES, BRAZIL
SEPTEMBER, 2017

ML2: AN EXPRESSIVE MULTI-LEVEL CONCEPTUAL MODELING LANGUAGE

CLAUDENIR MORAIS FONSECA

Dissertação submetida ao programa de Pós-Graduação em Informática do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para a obtenção do Grau de Mestre em Informática, sob orientação do Prof. Dr. João Paulo Andrade Almeida.

Banca Examinadora:

Universidade Federal do Espírito Santo

September, 2017.

Acknowledgments

To God, I am grateful above all things because He loved me and gave me salvation beyond all comprehension.

I would like to thank my supervisor, Prof. João Paulo Andrade Almeida, who guided me through so many years, teaching me a lot with words, but even more with examples. His competence, working ethics and kindness will be standards for me as professional and person. I would also like to thank Prof. Victorio Albani Carvalho and Prof. Julio Cesar Nardi, colleagues and co-supervisors that I had the opportunity to work with during my development as a student. These three I am happy to hold as special friends of mine.

I thank my colleagues of Nemo, my professors and the staff of the Federal University of Espírito Santo, who shared this place with me and helped to create this wonderful environment where I was able to grow and become who I am now.

I believe the path of a student must be humble, because everything that I know and everything that I have accomplished would not be if I were alone. Therefore I am thankful to everyone who shared this path with me, even for a brief moment.

I am thankful to my friends, who I hold dear, for all their support and encouragement.

I thank my family, my parents, Franklin and Selma, my siblings, Claudia and Clauber, and my extended family, Otoniel and Andrea. They have supported me in every way, allowing me to grow and seek the opportunities they were never granted. In every battle and every victory I carry their efforts, their tears and their prayers.

Finally, I thank my wife, Danielly, who is everything in my life, who has experienced all moments of my journey and never hesitated to support me, and love, and encourage me. Without her I would never be who I am, for she is part of me. To her I offer my love and my life.

Abstract

Subject domains are often conceptualized with entities stratified into a rigid two-level structure: a level of classes and a level of individuals which instantiate these classes. Multi-level modeling extends the conventional two-level classification scheme by admitting classes that are also instances of other classes, a feature which can be used beneficially in a number of domains. Despite the advances in multi-level modeling in the last decade, a number of requirements arising from representation needs in subject domains with multiple levels of classification have not yet been addressed in current modeling approaches. In this work, we investigate the requirements for multi-level modeling and propose an expressive multi-level conceptual modeling language dubbed ML2. We follow here a systematic approach based on a strict separation of concerns. First, we capture and formalize the conceptualization underlying multi-level modeling phenomena, called MLT*, building on the multi-level theory called MLT. Second, we employ MLT* as bedrock for the definition of ML2, a textual modeling language that addresses the elicited requirements for multi-level modeling. The proposed language is supported by a featured Eclipse-based workbench which verifies adherence of the ML2 model to the MLT* rules. The capabilities of ML2 are demonstrated by using it to accomplish three distinct modeling tasks: modeling a multi-level challenge proposed in the context of the MULTI 2017 workshop; modeling the concepts from ML2's underlying theory, MLT*; modeling the Unified Foundation Ontology (UFO).

Resumo

Domínios de interesse são muitas vezes conceituados com entidades estratificadas em uma estrutura rígida de dois níveis: um nível de classes e um nível de indivíduos que instanciam essas classes. A modelagem multi-nível estende o esquema convencional de classificação em dois níveis ao admitir classes que são também instâncias de outras classes, uma característica que pode ser empregada beneficamente em diversos domínios. Apesar dos avanços em modelagem multi-nível na última década, uma série de requisitos decorrentes da necessidade de representação de domínios de interesse com múltiplos níveis de classificação ainda não foram abordados pelas técnicas atuais. Neste trabalho, nós investigamos os requisitos para modelagem multi-nível e propomos um linguagem expressiva de modelagem conceitual multi-nível chamada ML2. Nós seguimos aqui uma abordagem sistemática baseada em uma separação estrita de interesses. Primeiramente, em uma teoria lógica denominada MLT*, capturamos e formalizamos a conceituação subjacente à modelagem de fenômenos que envolvam classificação em vários níveis. Esta teoria é uma extensão da teoria multi-nível chamada MLT. Em seguida, empregamos MLT* como alicerce na definição de ML2, uma linguagem textual de modelagem que atende aos requisitos elicitados para modelagem multi-nível. A linguagem proposta é apoiada por um *workbench* baseado em Eclipse que verifica a aderência de modelos ML2 às regras de MLT*. A efetividade de ML2 é demonstrada através de sua aplicação na realização de três tarefas distintas de modelagem: a modelagem de um desafio multi-nível proposto no contexto do workshop MULTI 2017; a modelagem dos conceitos da teoria subjacente à ML2, MLT*; e a modelagem da Unified Foundation Ontology (UFO).

List of Figures

List of Tables

List of Acronyms

MLT – Multi-level Theory

OCA – Orthogonal Classification Architecture

OMG – Object Management Group

OWL – Ontology Web Language

UFO – Unified Foundational Ontology

UML – Unified Modeling Language

W3C – World Wide Web Consortium

Contents

Chapter 1. Introduction

In this chapter, we present an overview of this work. We introduce multi-level modeling briefly and motivate the work by highlighting targeted issues in the research field (Section 1.1). We then define our objectives (Section 1.2), approach (Section 1.3), and describe the structure of this dissertation (Section 1.4).

1.1 Context and Motivation

A class (or type) is a ubiquitous notion in modern conceptual modeling approaches and is used in a conceptual model to establish invariant features of the entities in a domain of interest. Often, subject domains are conceptualized with entities stratified into a rigid two-level structure: a level of classes and a level of individuals which instantiate these classes. In many subject domains, however, classes themselves may also be subject to categorization, resulting in classes of classes (or metaclasses). For instance, consider the domain of biological taxonomies (Mayr, 1982; Brasileiro *et al.*, 2016b; Carvalho and Almeida, 2016). In this domain, a given *organism* is classified into *taxa* (such as, e.g., *Animal*, *Mammal*, *Carnivoran*, *Lion*), each of which is classified by a *biological taxonomic rank* (e.g., *Kingdom*, *Class*, *Order*, *Species*). Thus, to represent the knowledge underlying this domain, one needs to represent entities at different (but nonetheless related) classification levels. For example, *Cecil* (the lion killed in the Hwange National Park in Zimbabwe in 2015) is an instance of *Lion*, which is an instance of *Species*. *Species,* in its turn, is an instance of *Taxonomic Rank.* Other examples of multiple classification levels come from domains such as software development (Gonzalez-Perez and Henderson-Sellers, 2006) and product types (Neumayr, Grün and Schrefl, 2009).

An example of an early approach aiming at representing domains with these characteristics is the powertype pattern (Cardelli, 1988; Odell, 1994). In this pattern, instances of a type (the so-called "powertype") are specializations of a lower-level type (the so-called "basetype"). It is found regularly in many catalogues of modeling best practices, in which it appears as an ingredient of other patterns (see, for instance, (Fowler, 1996)). Given its importance in practice, it was also incorporated into the Unified Modeling Language (UML) (OMG, 2011), thus allowing modelers to specify a powertype in the context of a "generalization set". Despite the usefulness of this pattern, instantiation of powertypes is represented as a regular association between a powertype and the basetype, and is not given a specialized semantics (Carvalho, Almeida and Guizzardi, 2016). Further, models based on the pattern fail to capture fully the dual type/instance nature of domain elements. For example, instances of powertypes (unlike objects) cannot have values assigned to their attributes.

In the last decades, several approaches for the representation of multi-level models have been worked out, including those mostly focused on multi-level modeling from a model-driven engineering perspective (e.g., (Lara *et al.*, 2013; Frank, 2014)) and those that propose modeling languages for models with multiple levels of classification (e.g., (de Lara and Guerra, 2010; Atkinson and Gerbig, 2012)). These approaches embody conceptual notions that are key to the representation of multi-level models, such as the existence of entities that are simultaneously types and instances (classes and objects), the iterated application of instantiation across an arbitrary number of (meta)levels, the possibility of defining and assigning values to attributes at the various type levels, etc.

Despite these advances, a number of requirements arising from representation needs in subject domains have not yet been addressed in current modeling approaches. For example, in the aforementioned biology domain, we could be interested in the representation of a *discoverer* relation between instances of *Species* and instances of *Person* in order to identify the person who discovered a determined species. Many approaches cannot fully accommodate the representation of domain relations between elements of different classification levels (Carvalho and Almeida, 2016), which is the case here since *Species* classify types of individuals and *Person* classify individuals. Other approaches impose rigid constraints on the organization of elements into levels that obstruct the representation of genuine multi-level phenomena (Almeida, Fonseca and Carvalho, 2017); this makes it impossible to represent some general types such as “Thing” and “Type”, which are recurrent model elements whose instances span across different classification levels. Cases such as these suggest that a novel approach is required to address a broad set of requirements for multi-level models.

1.2 Objectives

To tackle the aforementioned issues, the goal of this work is to define an expressive conceptual modeling language for multi-level domains dubbed Multi-Level Modeling Language (ML2). The language uses as basis a well-founded multi-level theory dubbed MLT* and is designed to support a comprehensive set of representation requirements. From this goal, we can list a series of specific objectives that guide the progress of this work:

O1. Definition of requirements for a multi-level language that focuses on the representation needs for conceptual modeling;

O2. Development of a language-independent reference theory to serve as a semantic foundation for multi-level models;

O3. Specification of a multi-level modeling language that reflects the reference theory and addresses identified requirements ;

O4. Implementation of a modeling environment for the proposed language in order to support its practical application;

O5. Application of the language to selected subject domains.

1.3 Approach

We follow here a systematic approach based on a strict separation of concerns: first, the conceptualization underlying multi-level phenomena is captured by a reference theory; second, a modeling language that reflects the conceptualization captured by the reference theory is devised, while addressing technological and pragmatic concerns. This separation of concerns follows the view presented by Guizzardi (2005). In this view, a reference theory should be primarily shaped by the phenomena of interest, reflecting in the best way possible a certain vision of the world. Informing the design of a language with such a reference theory contributes to what is called the "domain adequacy" of the language, i.e. how closely a language is able to capture a certain domain of inquiry (Guizzardi, 2005).

In order to drive the development of a suitable solution for capturing multi-level domains, we investigate related works on conceptual modeling (e.g., (Odell, 1994; de Lara and Guerra, 2010; Atkinson and Gerbig, 2012)) and ontology engineering (e.g., (Masolo *et al.*, 2003; Foxvog, 2005; Guizzardi, 2005), and define a set of requirements for multi-level conceptual modeling (addressing specific objective O1).

Further, we develop a theory in first-order logics, dubbed MLT*, which satisfies the aforementioned requirements (addressing O2). Both requirements (Brasileiro *et al.*, 2016b) and theory (Carvalho and Almeida, 2016) here are improvements on the original contributions to the Multi-Level Theory (MLT). We advance the original works by adding requirements that account for more general subject domains and executing the necessary modifications on the theory. These modifications make it possible for ML2 to handle models with very generic concepts, such as "Entity" and "Thing", some of which are pervasive in ontology development (Masolo *et al.*, 2003; Foxvog, 2005; Guizzardi, 2005). Like MLT, MLT* is formalized through a light-weight formal technique, Alloy (Jackson, 2006), that allows testing (validation) and simulation of the formalization of a theory.

Moreover, we employ MLT* as the theoretical foundation for developing the Multi-Level Modeling Language, ML2. ML2 incorporates the definitions from MLT* on its constructs, allowing the specification of MLT* based models (addressing O3). Semantically-motivated syntactic rules for ML2 are provided, reflecting the rules of MLT*. We opt for a textual language (and a corresponding UML profile for visualization). Additionally, an Eclipse-based workbench is implemented through the Xtext[1] framework (addressing O4) in order to provide a proper environment for model development. The implementation of the ML2 Editor provides useful productivity tools, such as text highlighting, auto-completion, and model validation, which verifies the constraints of the language ruling out model that present inconsistences according to MLT*. The importance of model validation is highlighted by Brasileiro *et al.* (2016a), who found evidences that inconsistent multi-level models in the Semantic Web could be prevented by using such mechanisms.

We present ML2's capabilities by using it to model three distinct conceptual domains (addressing O5). First, we model a multi-level challenge proposed in the context of the MULTI 2017 workshop[2]. The challenge consists in a domain on product configurations, and, since it was developed independently, prevents bias in the selection of a domain for illustration of the technique. Second, we use ML2 to model ML2's underlying theory, MLT*. This example serves to show how ML2 is capable of dealing with quite general notions including those very concepts underlying the language. Finally, we model a fragment of the Unified Foundational Ontology (UFO) (Guizzardi, 2005). Through this last case, we show that ML2 can support a hierarchical approach for ontology-based multi-level conceptual models (Carvalho *et al.*, 2015), with models at varying levels of generality and domain specificity.

1.4 Structure

This dissertation is structured as follows: Chapter 2 contains a brief introduction to the subject of multi-level modeling and identifies requirements for a multi-level modeling language; Chapter 3 presents the multi-level theory MLT*, which serves as basis for the interpretation of multi-level models; Chapter 4 proposes the ML2 multi-level modeling language and compares it to others approaches present in the literature; Chapter 5 demonstrates the capabilities of ML2 by employing it on the specification of three distinct conceptualizations; finally, Chapter 6 presents the concluding remarks of this work.

[1] See https://eclipse.org/Xtext/.
[2] See https://www.wi-inf.uni-duisburg-essen.de/MULTI2017/.

Chapter 2. Multi-Level Modeling

Since early seventies, models revolutionized computer science by framing software information in useful abstractions, improving comprehension, documentation and communication (Chen, 1976). In this context, conceptual modeling emerged as modeling communities perceived the importance of capturing real-world information underlying a subject domain. In addition to promoting the representation of real-world domains, conceptual modeling has a focus on the formality of the representation means (Mylopoulos, 1992) relying on theoretical systems, such as logics, for supporting it.

Most modern conceptual modeling languages use the notion of type (or class) to capture invariant aspects of subject domains. In such languages, types are entities that classify other entities, namely instances, grouping them according to common features they have. Many modeling languages maintain a clear-cut division of the entities they describe into the categories of types and instances. This is a problem for domains in which the classification of types is required, leading to types whose instances are also types and breaking the dichotomy between instances and classifiers. For example, as discussed by Carvalho and Almeida (2016), considering the software development domain (Gonzalez-Perez and Henderson-Sellers, 2006) as discussed in (Carvalho and Almeida, 2016), project managers often need to plan according to the types of tasks to be executed during the development of software projects (e.g. “requirements specification”, “coding”). They may also need to classify those types of tasks giving rise to types of types of tasks. In this case, “requirements specification” and “coding” could be considered as examples of “technical task types”, as opposed to “management task types”. Finally, during project development, they need to track the execution of individual tasks (e.g. specifying the requirements of the system X). Thus, to describe the conceptualization underlying the software development domain, one need to represent entities of different (but nonetheless related) classification levels, such as tasks (specific individual occurrences), types of tasks, and types of types of tasks, leading to the development of multi-level models.

In the literature, two dominant kinds of approaches appear as solutions for modeling multi-level domains. The earliest kind consists of the powertype-based approaches (Cardelli, 1988; Odell, 1994), which use relations, other than instantiation, to represent types that classify other types. More recently, clabject-based approaches following (Atkinson and Kühne, 2003) emerged in revisiting the boundary between types and instances, proposing the notion of clabject as a type that could be considered instance of other types and present instance-like traits. We present these kinds of approaches in Section 2.1 and Section 2.2 respectively. In Section 2.3, we present the MLT Multi-Level Theory that is capable of harmonizing the two kinds of approaches, showing that there is no inconsistency in their combination.

The analysis of these different approaches along with the domains they are required to represent lead to the identification of a list of requirements for a multi-level language, presented in Section 2.4. Finally, in Section 2.5, we show how a number of existing multi-level approaches respond to the proposed list of requirements.

2.1 Powertypes

An early approach for dealing with domains that spam across different classification levels is the *powertype*. This approach was intended as a form of dealing with these domains in traditional two-level languages, i.e., languages that have a dichotomy of classes and instances. Following (Carvalho and Almeida, 2016), we take in consideration two main definitions of powertype present in the literature, (Cardelli, 1988) and (Odell, 1994).

Cardelli (1988) focus on a formal definition of powertype in analogy to the concept of *power set* from set theory (Bagaria, 2017). Considering a set “A”, the power set Power(A) is a set that contains all possible subsets of “A”, including “A” itself. Analogously, the powertype of a type “T” (also known as *basetype*) is the one whose instances are all possible subtypes of “T”, including “T” itself.

In contrast to (Cardelli, 1988), Odell (1994) proposes an informal definition of powertype focusing on its practical application in modeling languages. Odell aims at providing representation of powertypes in object-oriented modeling languages, being the most referenced notion of powertype in software engineering. His definition is simpler than Cardelli’s and states that a powertype is a type whose instances are specializations of another type. Take for example Figure 1, which shows a paradigmatic application of the powertype pattern: the instances of Tree Species are specific types of Tree, such as Sugar Maple and Apricot, and thus Tree Species is a powertype of Tree, a base type. The labeled generalization set (referred to as a subtype partition) identifies which specializations of “Tree” are instances of “Tree Species”. The current version of UML (OMG, 2011) gives support to Odell’s proposal for powertype representation in modeling languages.

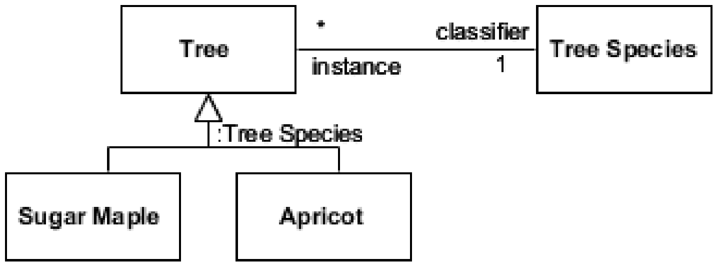


Figure 1 - Illustrating the notation proposed by Odell to associate *powertypes* with subtypes partitions (adapted from (Odell, 1994))

The powertype pattern cannot accommodate scenarios in which we are required to capture properties of types, e.g., the “average size” of instances of “Tree Species”. In this case, “Sugar

Maple” and “Apricot” cannot assign values to properties of “Tree Species” since they are types. These scenarios require powertype-based approaches to maintain instances of “Tree Species” to capture the instance facet of “Sugar Maple” and “Apricot”, therefore demanding the usage of two elements to represent a single domain concept.

2.2 Clabjects and Deep-Instantiation

As previously discussed in Section 2.1, the usage of powertypes allows the representation of types that classify types (or classes). This concept was originally employed in languages that obey a rigid two-level classification scheme (with a level of types and a level of instances), such as UML. Unlike those that have proposed the powertype pattern, Atkinson and Kühne (2000) propose a concept that acknowledges the duality of types in multi-level domains from the beginning, using the notion of *clabject*. A clabject is an entity that is both a type (or class) and an instance (object). As a type, a clabject defines properties for its instances. As an instance, it defines values of the properties of the type(s) it instantiates. Clabjects are often organized into levels where the entities at level M_n can instantiate entities at the level immediately above, M_{n+1}. Atkinson and Kühne refer to this organization of clabjects into adjacent levels as the *strict metamodeling principle*.

Figure 2 present an example of the usage of clabjects in a modeling tool called *Melanee* (Atkinson and Gerbig, 2012). In this example, “myMonitor” is an object that instantiates “Dell E1913”, which is in turn is a clabject that instantiates “Monitor Model”. In addition to the usage of adjacent levels, Atkinson and Kühne (2000) also propose the usage of “potencies” as indexes representing how many times an entity can be instantiated. Potency is a natural number that is decreased by one at each instantiation of a clabject, with the potency zero representing entities that cannot be instantiated. In Figure 2, “Monitor Model” has potency 2, since it can be instantiated into particular monitor models (e.g., “Dell E1913”), which can in turn be instantiated into particular monitors (e.g., “myMonitor”).

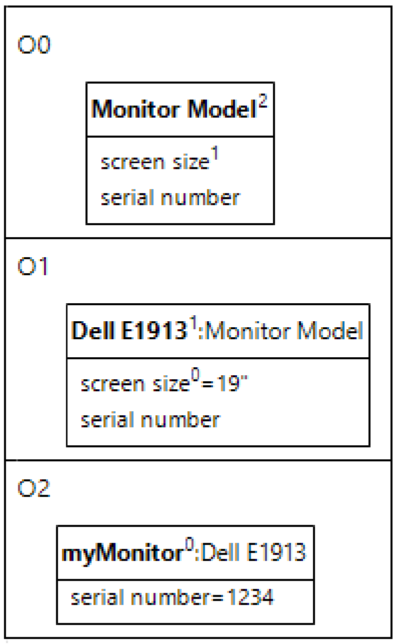


Figure 2 – Illustrating the usage of clabject into a modeling tool called *Melanee* (Atkinson and Gerbig, 2012).

Atkinson and Kühne also observed that, in multi-level domains, the features of a type may not only describe characteristics of its direct instances, but also have consequences on entities on more than one level below (Atkinson and Kuhne, 2001; Atkinson and Kühne, 2008). In order to accommodate this trait of multi-level domains, they propose *deep instantiation* in contrast to what they call *shallow instantiation*, which is the interpretation of general object-oriented approaches where types define properties that only impact their direct instances.

By also allowing the definition of potencies in attributes and associations, Melanee (Atkinson and Gerbig, 2012) incorporates support for deep instantiation. In Figure 2, the attribute “screen size” has potency 1, so it can only be instantiated and be assigned a value at the level immediately bellow, such as for “Dell E1913”. When the potency of an attribute is omitted, it is the same of the containing class, therefore “serial number” has potency 2 when declared at “Monitor Model”, potency 1 when instantiated at “Dell E1913”, and potency 0 when instantiated at “myMonitor”. Notice that the potency of an attribute or an association will never be greater than the potency of the class, as it cannot be further instantiated once the class potency reaches 0.

The notions of clabject and deep-instantiation, here presented for Melanee (Atkinson and Gerbig, 2012), also drive the implementation of other approaches in multi-level modeling, such as MetaDepth (Neumayr *et al.*, 2014) and DeepJava (Kuehne and Schreiber, 2007). Despite sharing main concepts, the particularities of each of these approaches are presented in Section 2.5.

2.3 MLT: The Multi-Level Theory

The third key approach for multi-level modeling we present here is the Multi-Level Theory (MLT), proposed by Carvalho and Almeida (2016). MLT is a system of axioms in first-order logics that aims at providing a foundation for multi-level domains relying solely on the *instantiation relation*, as a relation that may occur between two entities of subject domain. By using this relation, MLT is able to differentiate *types* and *individuals* as entities that, respectively, may or may not have instances. Here, the definition of type accommodates whatever entity that have others as instances, be it individuals or also types. In Figure 3, only “John”, “Bob” and “Ana” are individuals, while the rest are types. A characteristic of MLT is that, through the development of these very basic concepts, it is able to harmonize powertype-based and clabject-based approaches. Observe that the notation employed in Figure 3 and Figure 4, largely inspired in UML, is intended for purpose of example illustration only and does not suggest a syntax for MLT models.

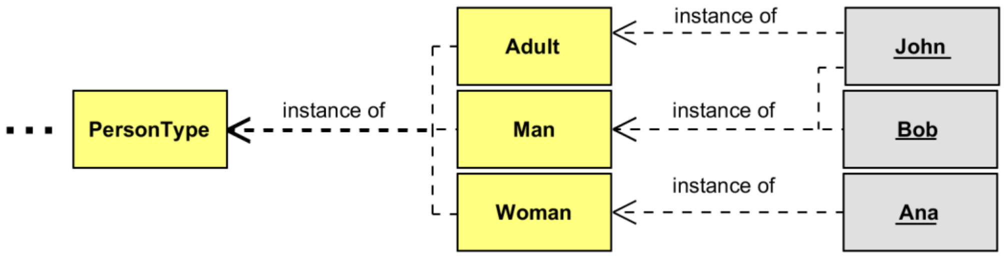


Figure 3 – Types and individuals in MLT.

MLT defines what it calls *structural relations*, relations derived from the instantiation relation used for capturing interactions between domain entities. Table 1 presents the most important structural relations of MLT. *Specialization* has a similar semantics to what is usually employed in conceptual modeling, where a type *t specializes* a type *t’* iff every instance of *t* also instantiates *t’*. This definition of specialization is not suitable for certain domain descriptions since it considers every type as specialization of itself. The *proper specialization* relation is defined with a more distinct notion of specialization where a type *t proper specialization* a type *t’* iff *t specializes t’* and *t* and *t’* are different entities. Through the *powertype* and *categorization* relations, MLT incorporates the notions of powertype of Cardelli (1988) and Odell (1994), respectively. The *categorization* relation is further refined into *complete categorization*, *disjoint categorization* and *partitions* considering whether the instances of the basetype are classified by at least one, at most one, or exactly one instance of the categorizer type, respectively.

Table 1 –MLT structural relations.

Structural Relation	Semantics
specializes(t',t")	A type t' specializes a type t" iff every instance of t' instantiates t".
properSpecializes(t',t")	A type t' proper specializes a type t" iff t' and t" are different types and every instance of t' instantiates t".
isPowertypeOf(t',t")	A type t' is powertype of a type t" iff every specialization of t" is instance of t'.
categorizes(t',t")	A type t' categorizes a type t" iff every instance of t' proper specializes t".
completelyCategorizes(t',t")	A type t' completely categorizes a type t" iff t' categorizes t" and every instance of t" instantiates at least one instance of t'.
disjointlyCategorizes(t',t")	A type t' disjointly categorizes a type t" iff t' categorizes t" and every instance of t" instantiates at most one instance of t'.
partitions(t',t")	A type t' partitions a type t" iff t' completely and disjointly categorizes t".

Moreover, MLT devises a pattern of model entities in order to account for the notion of levels, or *type orders*. The pattern consists of defining a more general type for each type order which in turn has as instances all possible entities at the order below. We can build this pattern from bottom-up: the type "Individual" has as instances all possible individuals, i.e., entities that cannot be instantiated (e.g., "John", the lion "Cecil", the dog "Lasie"); the type "First-Order Type" has as instances all possible types whose instances are individuals (e.g., "Person", "Lion", "Dog"); the type "Second-Order Type" has as instances all possible types whose instances are first-order types (e.g., "Species"), and so on. As a consequence of this definition, every type within a type order specializes the *basic type* (e.g., "Individual", "First-Order Type", "Second-Order Type") of that order and instantiates the *basic type* of the order above, what characterizes the existence of powertype relations between *basic types* of adjacent orders (see Figure 4). This pattern of *basic types* can be extended as required, serving as foundation for definition of types in orders, similar to the usage of potencies to clarify the level of clabject. Returning to the example of "Monitor Model", "Dell E1913" and "myMonitor", "myMonitor" is an instance of "Individual", "Dell E1913" an instance of "First-Order Type" and specialization of "Individual", and "Dell E1913" an instance of "Second-Order Type" and specialization of "First-Order Type".

In addition to the relations and model patterns proposed in the theory that harmonize the powertype and clabject approaches, MLT also goes beyond *shallow instantiation* by proposing the notion of *regularity features*. Differently from deep instantiation, which allows any number of repetitions of an attribute in an instantiation chain, regularities admit features that have impact over one more instantiation level. In other words, a feature may regulate the assignments of features of types at the level below. Considering the example from Figure 4, the attribute "instancesScreenSize" of "MobilePhoneModel" regulates the value of "screenSize" of "MobilePhone". When assigned, the attribute "instancesScreenSize" determines the values of "screenSize" for all instances of "IPhone5". Likewise, the attributes "instancesMinStorageCapacity" and "instancesMaxStorageCapacity" determines the range of "storageCapacity" for mobile phones, which is between 16GB and 32GB for instances of "IPhone5". Regularity features are applicable to both attributes and associations in MLT.

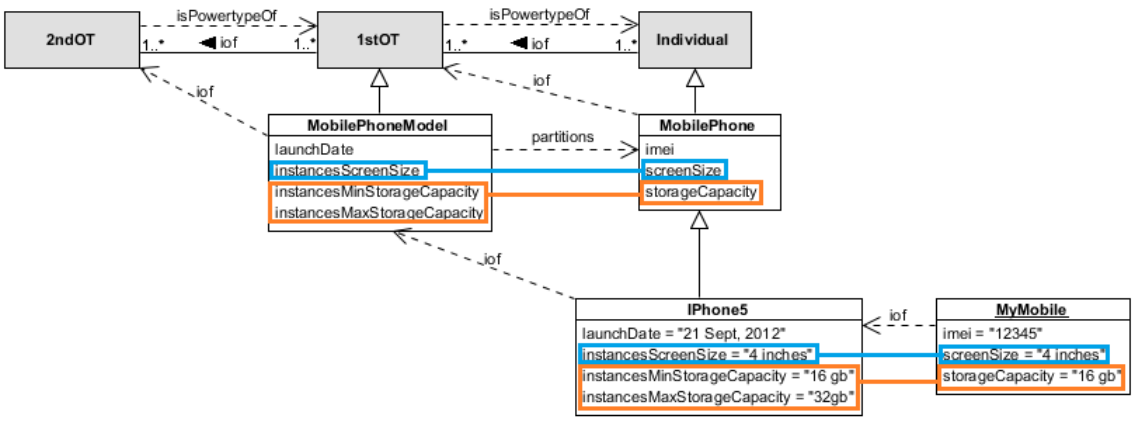


Figure 4 – Illustrating the notion of *regularity features* (Carvalho and Almeida, 2016).

2.4 Requirements for Multi-Level Conceptual Modeling Languages

We establish here key requirements for a multi-level conceptual modeling language, substantiating these requirements with sources from the literature on multi-level modeling and justifying them based on intended usage scenarios (i.e., representation needs). The set of requirements discussed here serves later as the basis for a comparison of our approach with existing multi-level modeling approaches. We separate this set as follows: Section 2.4.1 presents the requirements related to the capacity of capturing the entities from the conceptual domain; Section 2.4.2 presents the requirements related to modeling features of entities from the conceptual domain in a multi-level context.

2.4.1 From a Two-Level to a Multi-Level Scheme

First of all, given the nature of a multi-level scheme, an essential requirement for a multi-level modeling language is *the ability to represent entities of multiple (related) classification levels*, capturing chains of instantiation between the involved entities (requirement **R1**). To comply with this requirement, the language must admit entities are simultaneously types (class) and instances (object) (Atkinson and Kühne, 2000), shown in Figure 3 where "Adult", "Man" and "Woman" are both classified by "PersonType" and classifiers of "John", "Bob" and "Ana". This means that a multi-level language differs from the traditional two-level scheme, in which classification (instantiation) relations can only be established between classes and individuals.

The size of these chains of instantiation in a conceptual model may vary according to the nature of the phenomena being captured and according to the model purposes. Because of this, *a general-purpose multi-level modeling language shall allow the representation of an arbitrary* *number of classification levels* (**R2**) (including the two-level scheme as a special case). The ability to deal with an arbitrary number of levels is pointed out by authors such as (Frank, 2014) and (Atkinson and Gerbig, 2012), as a key a requirement for multi-level modeling approaches. Several examples of three and four level models are available in the literature, as well as in structured data repositories such as Wikidata (in which there are more than 17,000 classes involved in multi-level taxonomies (Brasileiro *et al.*, 2016a)).

Further, there is empirical evidence to support the claim that representations capturing chains of instantiation can benefit greatly from *principles* to guide the organization of entities into levels. It was found that, without proper support, multi-level taxonomies are built in an unsound way (Brasileiro *et al.*, 2016a), for example, due to the inadequate use of instantiation (and its combination with subtyping). An example of such is presented in Figure 5, where "Computer Scientist" both instantiates and specializes (by transitivity) "Profession". Therefore, according to this model "Tim Berners-Lee" is an instance of "Profession", a clear violation of the concept "Profession".

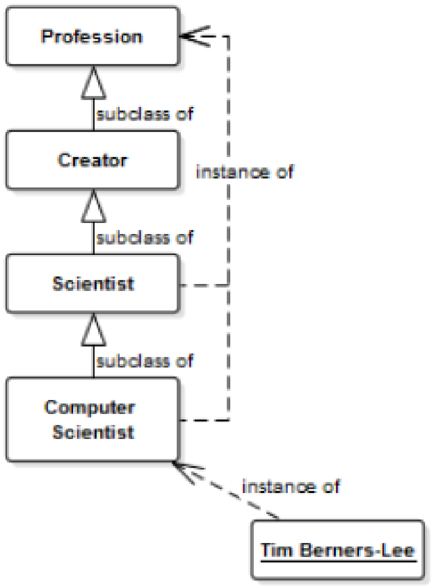


Figure 5 – Example of inconsistent model found on Wikidata (Brasileiro *et al.*, 2016a)

In fact, over 87% of the classes in multi-level taxonomies in Wikidata were involved in errors that could have been prevented with guidance from the editing/modeling environment (Brasileiro *et al.*, 2016a). Based on this evidence, we consider that *a multi-level modeling language shall define guiding principles for the organization of entities into levels* (**R3**). These principles should guide the modeler on the adequate use of classification (instantiation) relations. The strict metamodeling principle (Atkinson and Kühne, 2000), which prescribes the arrangement of elements into levels, is an example of solution that fulfills this requirement.

While these principles are intended to guide the modeler in producing sound models, they should not obstruct the representation of genuine multi-level phenomena. The strict metamodeling principle, for example, excludes from the domain of enquiry abstract notions such as a universal “Type” or, an even more abstract notion such as “Thing”. This is because their instances may be related in chains of instantiation, conflicting with the stratification imposed by the guiding principle. Given that these general notions are ubiquitous in comprehensive conceptualizations (see e.g., the core of the Semantic Web with the notion of “Resource” or “Thing” (W3C, 2009, 2014), (foundational) ontologies such as UFO (Guizzardi, 2005), Cyc (Foxvog, 2005), DOLCE and BFO (Masolo *et al.*, 2003) with their notions of “Entity” or “Thing”, Telos (Mylopoulos *et al.*, 1990) with the notions of “Property”), we conclude that *an expressive multi-level modeling language should support the representation of types that defy a strictly stratified classification scheme* (**R4**) (with the general notion of “type” or “class” and the universal notion of “entity” or “thing” as paradigmatic special cases).

Finally, an important characteristic of domains spanning multiple levels of classification is that there are domain rules that apply to the instantiation of types of different levels, leading to the necessity of representing ‘structural relations’ that govern the instantiation of types of different levels. For example, all instances of “Dog Breed” (e.g. “Collie” and “Beagle”) specialize the base type “Dog”. In order to represent “Dog Breed”, it is, thus, key to establish its relation with the “Dog” type. Further, in this case, to clarify the modeling intent, one should represent whether an instance of “Dog” may instantiate: (i) only one, or (ii) more than one “Dog Breed”. The powertype pattern (Cardelli, 1988; Odell, 1994) is an example of solution based on this notion, where the identification of the relation between the powertype (“Dog Breed”) and the basetype (“Dog”) is necessary. We conclude thus that, *an expressive multi-level modeling language should be able to represent rules that govern the instantiation of related types at different levels* (**R5**) (supporting the *powertype pattern* as a special case, given its importance in several application domains (Lara, Guerra and Cuadrado, 2014)).

2.4.2 Relations and Attributes in a Multi-Level Scheme

Types can be regarded as entities that capture common features of other entities that are considered their instances. These features are often captured using the notions of *attributes* and *relationships* (Chen, 1976). In a two level scheme, features are only defined at type level and given values at object level. In a multi-level scheme, however, features may be defined at a (higher) level and assigned values at another (lower) level. For example, consider a domain in which each “Cellphone Model” (as a type of “Cellphone”) is “designed by” a “Person”. An instance of “Cellphone Model” (a type such as “IPhone5”) is linked to a particular “Person” as its designer (such as “Jony Ivy”). *A multi-level modeling language should thus support the representation of features (attributes and relationships) of types as well as the assignment of values to their instances (regardless of whether they are themselves types or objects)* (**R6**).

Further, a recurrent phenomenon in domains dealing with multiple classification levels is that features of types in one classification level may constrain features in lower levels. For example, considering that every cellphone has a screen, we may define screen size as a feature that characterizes cellphones. Consider further that specific cellphone models prescribe a particular screen size. In this scenario, a feature of a cellphone model, such as the “IPhone5”, constrains features of individual cellphones, such as “John’s IPhone5” which is an instance of “IPhone5”. “John’s IPhone5” has a 4-inch screen, respecting the screen size defined for “IPhone5” as an instance of “CellphoneModel”. To be able to represent these phenomena, *an appropriate multi-level modeling language should include support to describe rules relating features of entities in different levels* (**R7**). The *deep instantiation* (Atkinson and Kühne, 2008) is an example of mechanism to capture a specific sort of relations between attributes of entities in different levels.

Finally, in various domains, there are relations that may occur between entities of different classification levels (Neumayr *et al.*, 2014). For example, consider the following domain rules: (i) each “Cellphone” has an owner (a “Person”), (ii) each “Cellphone” is classified as instance of a “Cellphone Model”, and (iii) each “Cellphone Model” is designed by a “Person”. In this domain, instances of “Person” (individuals) must be related simultaneously with instances of “Cellphone Model” (which are classes) and also with instances of “Cellphone” (which are individuals, in this case, instances of instances of “Cellphone Model”). Thus, *a multi-level modeling language should allow the representation of domain relations between entities of various classification levels* (**R8**).

2.5 Related Work

We discuss here a number of multi-level representation techniques reported in the literature, focusing on their satisfaction of the requirements defined in Section 2.4. In addition to various multi-level techniques (including DeepTelos, DeepJava, Melanee, M-Objects, MetaDepth, Kernel), we also discuss UML’s support for the powertype pattern, given its practical importance.

In the UML 2.4.1 specification (OMG, 2011), a class plays the role of “powertype” whenever it is connected to a generalization set composed by the generalizations that occur between a base classifier and the instances of the powertype. Given that generalization sets only exist when specializations of the base type are modeled, the UML cannot capture simple multi-level models in which instances of a powertype are omitted. As discussed in (Carvalho, Almeida and Guizzardi, 2016), this rules out simple models such as “DogBreed” categorizing “Dog”, when specific breeds are omitted. Hence, we consider the UML to only partially satisfy R1. In UML, chains of instantiation of arbitrary size can be captured by cascading the powertype pattern iteratively (again requiring the use of explicit specializations in generalization sets), thus partially satisfying R2. Further, the UML specification does not provide principles to guide the organization of entities into (classification) levels[3]. The only rule in UML concerning the consistency of instantiation chains aims at avoiding a "powertype" to be an instance of itself. Due to this incompleteness, it fails to satisfy R3. This very same constraint rules out some orderless types, such as the type "Type". Therefore, we consider that the UML only partially meets requirement R4. We consider that the notion of "powertype" in UML corresponds to MLT's notion of categorization, failing to capture Cardelli's powertype, since all instances of the powertype must be members of an identified generalization set. Thus, we consider that R5 is only partially met by UML[4]. Given that instances of powertypes cannot have values assigned to their features, UML fails to satisfy R6, and thus, R7 and R8.

DeepTelos is a knowledge representation language that approaches multi-level modeling with the application of the notion of "most general instance (MGI)" (Jeusfeld and Neumayr, 2016). The authors revisit the axiomatization of Telos (Jarke *et al.*, 1995) and add the notion of MGI to Telo's formal principles for instantiation, specialization, object naming and attribute definition. The notion of MGI can be seen as the opposite of Odell's powertype relation. For example, to capture that "Tree Species" is a "powertype" (in Odell's sense) of "Tree", in DeepTelos it would be stated that "Tree" is the "most general instance" (MGI) of "Tree Species". Considering the MGI construct allows the representation of entities in multiple classification levels and that DeepTelos allows the representation of chains of MGI to represent as many levels as necessary, we consider that DeepTelos meets R1 and R2. DeepTelos builds up on Telos, whose architecture defines the notions of *simple class* and *w-class* which are analogous to the notions of ordered and orderless types we have defined here. Nevertheless, stratification rules for simple classes (constraining specialization and cross-level relations) are not axiomatized. Thus, we consider that it partially meets R3 and that it meets R4 with the notion of *w-class*. Considering that DeepTelos provides only the concept of MGI to constrain the instantiation of types in different levels, not elaborating on the nuances of the relations between higher-order types and base types, we consider that it partially meets requirement R5. It supports the attribution of values to features of types, meeting R6. However, its account for attributes does not include any support to explain the relationship between attributes of entities in different classification levels, not meeting requirement R7. Finally, DeepTelos admits relations between types in different levels, thus, meeting requirement R8.

DeepJava is an extended version of Java that supports multi-level mechanisms for programming languages (Kuehne and Schreiber, 2007). The language allows the specification of potencies for Java classes and fields along with instantiation for classes. In DeepJava the potency of an element denotes the maximum depth of its instantiation chain, or how many times

[3] Note that, since our focus is on ontological instantiation, we are not addressing here OMG's fixed four-layer language architecture.

[4] The UML extension proposed in (Carvalho, Almeida and Guizzardi, 2016) meets R5 by providing constructs to represent all of the MLT's cross-level relations.

type can be instantiated. Through this mechanism, DeepJava is able to define entities at an arbitrary number of classification levels, defining the level on which each entity sits, thus satisfying requirements R1, R2 and R3. As the language only accounts for defined potencies with direct instantiation, it does not account for entities that defy the stratification in levels, not meeting requirement R4. Applying potencies in tandem with specializations, DeepJava allows representing that all instances of a (higher-order) type specialize another type. Considering that such mechanism maps Odell's notion of powertype (and MLT's notion of characterization), not elaborating on further nuances of the relations between higher-order types and base types, we consider that it partially meets requirement R5. As a programing language, DeepJava supports references between any objects in memory, and both feature specification and assignment at any classification level (except at the highest one, since a pure Java class does not have features to which values can be assigned). However, the only mechanism available for relating features across levels is potency, which is limited to define how deep a feature is present in an instantiation chain. Therefore, DeepJava meets requirements R6 and R8, but only partially meets R7.

Melanee (Atkinson and Gerbig, 2012) is a tool that supports multi-level modeling founded on the notions of *clabject* and *potency*. It is based on the idea of assigning to *clabjects* and *fields* (attributes and slots) a *potency*, which defines how deep the instantiation chain produced by that *clabject* or *field* can become. When a *clabject* is instantiated from another *clabject*, the potencies of the created *clabject* and of its *fields* are given by the original *clabject* and *fields* potencies decremented by one. Objects have potency equal to zero indicating they cannot be instantiated[5]. If the potency of a *field* becomes zero then a value can be assigned to that *field*. This mechanism allows Melanee to represent entities in multiple classification levels, organizing and capturing the instantiation chains allowing an arbitrary number of levels, thus, meeting requirements R1, R2 and R3. Melanee also defines the notion of *star potency* as a means to support the representation of types having instances of different potencies. While this allows for the representation of some types that defy stratification, star potency does not allow self-instantiation, which is required for the abstract types we have dealt with here. Therefore, we consider that it partially meets R4. In Melanee, instantiations are the only relations that may cross level boundaries and no constructs are provided to capture rules concerning instantiation at different levels (such as the cross-level relations of MLT). Therefore, we consider that it does not satisfy requirements R5 nor R8[6]. Melanee supports the attribution of values to features of types, thus, meeting R6. Finally, it meets R7 with a combination of the notions of attribute *durability* and *mutability* (Clark, Gonzalez-Perez and Henderson-Sellers, 2014).

5 In Melanee and other potency based approaches; zero potency is also used indistinctively to represent abstract classes.
6 The inclusion of MLT cross-level relations for Melanee to satisfy R5 is currently being investigated (in cooperation with Atkinson).

In (Neumayr, Grün and Schrefl, 2009) the authors propose a multi-level modeling approach founded on the notion of *m-object. M-objects* encapsulate different levels of abstraction that relate to a single domain concept, and an m-object can *concretize* another m-object. The *concretize relationship* comprises classification, generalization and aggregation relationships between the levels of an m-object (Neumayr, Grün and Schrefl, 2009). We observe that this is a semantic overload between three relationships of quite different ontological nature, which can affect the understandability and usability of the approach. Since the m-objects approach allows the representation of entities in an arbitrary number of levels relating them through chains of *concretize relationships*, we consider it meets requirements R1 and R2. Given that the approach adopts a stratified schema in which *concretize relationships* may only relate types in adjacent levels, we consider that it meets R3 and does not meet R4. Further, since the *concretize relationships* are the only structural relationships that cross level boundaries, the approach fails to meet R5. It provides support to represent features of types (meeting R6), but it does not include support to explain the relationship between attributes of entities in different classification levels (not meeting R7). Finally, in (Neumayr *et al.*, 2014), the authors observe that the approach was unable to capture certain scenarios in which there are domain relations between m-objects at different instantiation levels. To address this limitation, the approach was extended with the concept of Dual-Deep Instantiation, which allows the representation of relations between m-objects at different instantiation levels through the assignment of a potency to each association end, thereby satisfying R8.

MetaDepth is a textual multi-level modeling language founded on the same notions of clabject, potency, durability and star potency used by Melanee. Differently from Melanee, MetaDepth supports the representation of domain relationships as references, such that each reference has its own potency (a solution close to the one adopted in Dual-Deep Instantiation (Neumayr *et al.*, 2014)), allowing the representation of domain relations between clabjects at different instantiation levels. Therefore, MetaDepth meets the all the requirements Melanee does, and also succeeds on meeting requirement R8.

Kernel (Clark, Gonzalez-Perez and Henderson-Sellers, 2014) was proposed as a foundation for model-based language engineering. A Kernel “class” is also an “object” and, as such, it can instantiate other “classes” iteratively, thereby satisfying R1 and R2. It supports R4, R6, and R8, since it is rather unconstrained in order to support the definition of various multi-level modeling mechanisms. Given its focus as an agnostic basis, it does not aim at directly supporting organization principles, structural rules nor deep instantiation mechanisms (therefore it does not aim at supporting R3, R5 and R7). Nonetheless, this focus of Kernel allows it to describe others approaches, such as potency-based and powertype-based approaches.

The Open integrated framework for Multi-Level Modeling (OMLM) (Igamberdiev *et al.*, 2016) is a multi-level approach focused on a strict separation of concerns between three dimensions: the ontological dimension, concerned with the subject domain; the linguistic dimension, concerned with the linguistic elements involved in the representation of the domain; and the realization dimension, which focus on mapping models to a implementation target of choice. By making use of Flora-2 (Yang *et al.*, 2005), an F-Logic dialect, OMLM supports a clabject-based representation of multi-level domains, with the advantage of allowing the user to extend the language by adding constructs and syntactic rules. Originally, OMLM supports the representation of entities in multiple (unbound and related) classification levels, satisfying R1, R2 and R3. OMLM, however, in its ontological dimension, it does not support types that defy the organization of entities into levels and does not satisfy R4, solely allowing instantiation relations between adjacent levels. The language also fails to satisfy R5 as there are no others relations besides instantiation for guiding the classification of entities. Attributes in OMLM are considered *single-potency* elements, i.e., elements that can be instantiated only once in the ontological dimension. This treatment of attributes satisfies R6, but fails to satisfy R7 since there is no mechanism for supporting the representation of related features at different levels. In a previous version of OMLM called MiF (Igamberdiev, Grossmann and Stumptner, 2014), the same authors claim that their language does not support cross-level domain relations, not satisfying R8, even though can potentially be extended in that sense.

Finally, Selway *et al.* (2017) propose on their work the SLICER conceptual framework, which also accounts for multi-level models. SLICER provides to the user a set of level-aware relations that enable multi-level modeling, such as specializations, instantiations and powertype ("subset by specification") relations. In SLICER, not only instantiation characterizes the transition between "levels", but also specialization when properties are added to a super type. Some rules for levels are provided using these relations. SLICER is able to address R1 and R2 through the definition of entities in an unbound number of classification levels, but we consider it to partially address R3 since the rules for organization into levels are rather loose (despite being well defined). Despite that, the rules imposed on specialization and instantiation prevent some general types such as "Type" and "Thing" from being represented (e.g., because of self-instantiation) R4. SLICER is able to address R5 through the *Subset-by-Specification* relation, which has the same semantics of Odell's notion of powertype (Odell, 1994) and includes variations based on complete and disjoint constraints. At last, the language supports both *shallow* and *deep instantiation*, and does not impose constraints over domain relations between entities of different levels (addressing R6, R7 and R8).

Table 2 summarizes our evaluation of the various related approaches against the proposed requirements.

Table 2 – Requirements comparison among multi-level modeling languages.

Requirements	***UML***	***DeepTelos***	***DeepJava***	***Melanee***	***M-objects***	***MetaDepth***	***Kernel***	***OMLM***	***SLICER***
R1 – represents entities of multiple classification levels	Partially	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
R2 – arbitrary number of classification levels	Partially	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
R3 – defines guiding principles for organization of models	No	Partially	Yes	Yes	Yes	Yes	No	Yes	Partially
R4 – types that defy a stratified classification scheme	Partially	Yes	No	Partially	No	Partially	Yes	No	No
R5 – represent rules to govern instantiation of related types	Partially	Partially	Partially	No	No	No	No	No	Yes
R6 – represents features and feature assignments	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
R7 – relates features of entities in different levels	No	No	Partially	Yes	No	Yes	No	No	Yes
R8 - domain relations between entities in various levels	No	Yes	Yes	No	Yes	Yes	Yes	No	Yes

None of the approaches analyzed fully addresses the identified requirements for multi-level conceptual modeling, suggesting that a novel language is required. In Chapter 4, we present a modeling language called ML2 to address the identified demands of multi-level conceptual modeling. The theoretical basis of ML2 is built upon MLT, as it show a good adherence to the aforementioned requirements failing only to support types that defy organization into strictly stratified schemes.

Chapter 3. Theoretical Basis for Multi-Level Conceptual Modeling

In order to provide a theoretical foundation for multi-level conceptual modeling, Carvalho and Almeida (2016) have proposed a formal axiomatic theory called MLT, founded on the notion of instantiation. MLT has been used successfully to analyze and improve the UML support for modeling the powertype pattern (Carvalho, Almeida and Guizzardi, 2016), to uncover problems in multi-level taxonomies on the Web (Brasileiro *et al.*, 2016a), to found an OWL vocabulary that supports the representation of multi-level vocabularies in the Semantic Web (Brasileiro *et al.*, 2016b), and to provide conceptual foundations for dealing with types at different levels of classification both in core (Carvalho and Almeida, 2015) and in foundational ontologies (Carvalho *et al.*, 2015). As discussed in Chapter 2, MLT is unable to deal with types that defy strictly stratified schemes. This has motivated the development of an extended version of MLT, dubbed MLT* (Almeida, Fonseca and Carvalho, 2017). This chapter presents this theory, largely based on (Almeida, Fonseca and Carvalho, 2017).

3.1 Basic Notions

The notions of *type* and *individual* are central for our multi-level modeling theory. *Types* are predicative entities that can possibly be applied to a multitude of entities (including types themselves). Particular entities, which are not types, are considered *individuals*. Each type is characterized by an *intension, which* is used to judge whether the type applies to an entity (e.g., whether something is a Person, a Dog, or a Chair). It is also called *principle of application* in (Guizzardi, 2005). If the intension of a type *t* applies to an entity *e* then it is said that *e is an instance of t.* Thus, the *instance of* relation (or *instantiation* relation) maps a type to the entities that fall under the type. The set of instances of a type is called the *extension* of the type (Henderson-Sellers, 2012).

MLT* is formalized in first-order logic, quantifying over all possible individuals and types. The theory is built up from the instantiation relation, which is formally represented by a binary predicate iof(*e*,*t*) that holds if an entity *e* is instance of an entity *t* (denoting a type). For example, the proposition iof(John,Person) denotes the fact that “John” is an instance of the type “Person”.

Using the iof predicate, we can define the ground notion of *individual* (D1). An entity is an individual iff it does not possibly play the role of type in instantiation relations. Conversely, an entity is a type iff it plays the role of type in instantiation relations, i.e., if there is some entity which instantiates it (D2). Definitions D1 and D2 create a dichotomy with all elements in the domain of quantification being considered either types or individuals.

$$\forall x(\text{individual}(x) \leftrightarrow \neg\exists y(\text{iof}(y,x))) \quad \text{(D1)}$$

$$\forall x(\text{type}(x) \leftrightarrow \exists y(\text{iof}(y,x))) \quad \text{(D2)}$$

We assume that the theory is only concerned with types with non-trivially false intensions, i.e., the theory is only concerned with types that have possible instances in the scope of the conceptualization being considered. Therefore, we judge that types do not depend on the existence of instances whenever they exist, but in some possible world. For example, the type "Dinosaur" is still a type even if it has no current instances, and the type "Unicorn" is valid as long it has possible instances within some fictional conceptualization. This view of valid types is shared with others works on conceptual modeling, such as (Guizzardi, 2005).

We assume that all types are ultimately founded on individuals (A1) (not unlike well-founded set-theory, in which sets are ultimately founded or urelements which are themselves not sets). Thus the transitive closure of the instantiation relation (iof'), always leads us from a type to one or more individuals.

$$\forall t(\text{type}(t) \rightarrow \exists x(\text{individual}(x) \wedge \text{iof}'(x,t))) \quad \text{(A1)}$$

Note that the definitions so far allow us to satisfy **R1**, as we place no restrictions on the kinds of entities that may instantiate a type. Thus, the theory would admit a model such as the one illustrated in Figure 6. The figure depicts a chain of instantiation, with "Man" and "Woman" instantiating "PersonTypeByGender", and "John" and "Bob" instantiating "Man", while "Ana" instantiates "Woman". We use a notation inspired in the class and object notations of UML, and we use dashed arrows to represent relations that hold between the elements, with labels to denote the relation that applies (in this case *instance of*). This notation is used in all further diagrams in this chapter. It is important to highlight here that our focus is not on the syntax of a multi-level modeling language yet and we use these diagrams to illustrate the concepts intuitively. Our solution for syntax of a multi-level modeling language will be presented in later chapters of this work. Further, no constraint is placed on the size of instantiation chains, and thus, the theory would admit a model such as the one illustrated in Figure 7 (satisfying **R2**).

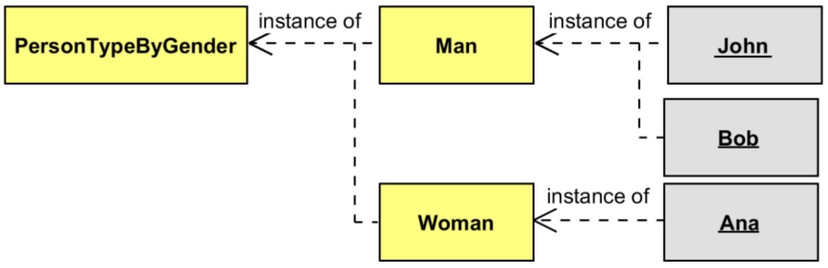


Figure 6 – An instantiation chain, where "Man" and "Woman" are both instances and classes.

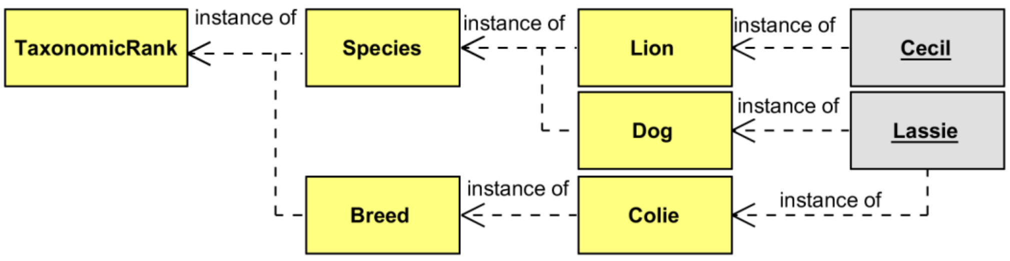


Figure 7 – A four-level instantiation chain with representing a biological domain.

We define some basic structural relations, starting with the ordinary specialization between types. In our theory, structural relations are relations that govern the instantiations among the instances of the relata. In the case of specialization, a type *t specializes* another type *t'* iff all possible instances of *t* are also instances of *t'*. According to this definition every type *specializes* itself. Since this may be undesired in some contexts, we define the *proper specialization* relation in which *t proper specializes t'* iff *t specializes t'* and *t* is different from *t'*.

$$\forall t_1, t_2 \text{ specializes}(t_1, t_2) \leftrightarrow \text{type}(t_1) \wedge \text{type}(t_2) \wedge \forall e(\text{iof}(e, t_1) \rightarrow \text{iof}(e, t_2)) \quad \text{(D3)}$$

$$\forall t_1, t_2 \text{ properSpecializes}(t_1, t_2) \leftrightarrow (\text{specializes}(t_1, t_2) \wedge \neg(t_1 = t_2)) \quad \text{(D4)}$$

We consider two types equal iff the sets of all their possible instances are the same[7]. This definition of equality only applies to elements which are not *individuals*, hence the 'guard' conditions on the left-hand side of the implication:

$$\forall t_1, t_2((\text{type}(t_1) \wedge \text{type}(t_2)) \rightarrow (t_1 = t_2) \leftrightarrow \forall x(\text{iof}(x, t_1) \leftrightarrow \text{iof}(x, t_2))) \quad \text{(D5)}$$

Building up on the specialization definition, we can now address the notion of powertype. Here we employ the seminal notion proposed by (Cardelli, 1988). According to Cardelli, the same way specializations are intuitively analogous to subsets, *powertypes* can be intuitively understood as powersets. The powerset of a set A is the set whose elements are **all** possible subsets of A including the empty set and A itself. Thus, "if A is a type, then Power(A) is the type whose elements are **all** the subtypes of A" (including A itself)(Cardelli, 1988). For example, in Figure 8 "PersonType" is the powertype of "Person", thus every specialization of "Person" (e.g. "Man", "Woman", "Adult" and even "Person" itself) instantiates it, regardless of how deep it is in a specialization hierarchy (e.g. "AdultMan" is also an instance of "PersonType").

[7] See (Carvalho and Almeida, 2016) for a refinement of identity and specialization concerning modal distinctions.

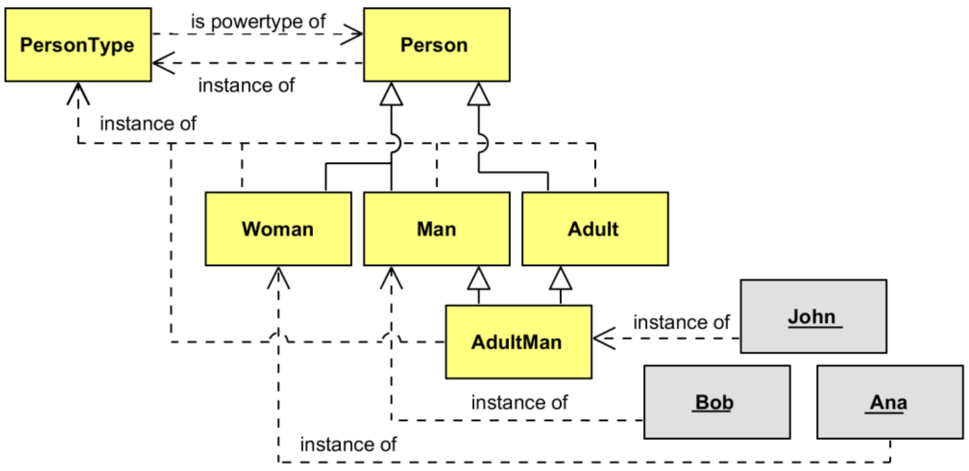


Figure 8 – PersonType and its instances

Following Cardelli's definition, we define that a type *t1 is powertype of* a type *t2* iff all *instances of t1* are *specializations of t2* and all possible *specializations of t2* are *instances of t1*. In this case, *t2* is said the base type of *t1*:

$$\forall t_1, t_2 \ \mathrm{isPowertypeOf}(t_1, t_2) \leftrightarrow \mathrm{type}(t_1) \wedge \forall t_3(\mathrm{iof}(t_3, t_1) \leftrightarrow \mathrm{specializes}(t_3, t_2)) \quad \text{(D6)}$$

Given the definition of powertype, it is possible to conclude that each type has at most one *powertype* (theorem T1) and that each *type* is *powertype of*, at most, one other type (theorem T2). These theorems are proved in (Carvalho and Almeida, 2016), which suggests a concrete syntactic constraint for a multi-level model: only one higher-order type can be linked to a base type through the *is powertype of* relation.

$$\forall p, t(\mathrm{isPowertypeOf}(p, t) \rightarrow \neg\exists p'((p \neq p') \wedge \mathrm{isPowertypeOf}(p', t))) \quad \text{(T1)}$$

$$\forall p, t(\mathrm{isPowertypeOf}(p, t) \rightarrow \neg\exists t'((t \neq t') \wedge \mathrm{isPowertypeOf}(p, t'))) \quad \text{(T2)}$$

3.2 Accounting for Stratification into Orders

Note that, thus far, the theory does not impose a principle of organization for the entities into (strictly stratified) 'levels'. In order to account for such kinds of principles of organization, we use the notion of type order. Types whose instances are individuals are called *first-order types*. Types whose instances are *first-order types* are called *second-order types*. Those types whose extensions are composed of *second-order types* are called *third-order types*, and so on.

Types that follow this strictly ordered scheme are called *ordered types*. To define such a scheme formally, we define a notion of 'basic type'. A basic type is the most abstract type in its type order. For example, "Individual" is a basic type since it is the most abstract of all first-order types, classifying all instances of first-order types, i.e., all possible individuals. We define the constant "Individual" as follows:

$$\forall t((t = \mathrm{Individual}) \leftrightarrow \forall x(\mathrm{individual}(x) \leftrightarrow \mathrm{iof}(x, t))) \quad \text{(A2)}$$

Like “Individual”, there are basic types for each subsequently higher order, i.e., every instance of the basic type of an order *i* (i>1) specialize the basic type of the order immediately below (i-1). This is formalized by D7. (Note that *i* is only used to improve the intuition in the definition, and is not formally a variable).

$$\forall b_i(\text{basictype}(b_i) \leftrightarrow$$
$$(\forall x(\text{individual}(x) \leftrightarrow \text{iof}(x, b_i)) \vee$$
$$\exists b_{i-1}(\text{basictype}(b_{i-1}) \wedge \forall t_{i-1}(\text{specialize}(t_{i-1}, b_{i-1}) \leftrightarrow \text{iof}(t_{i-1}, b_i))))) \quad \text{(D7)}$$

A consequence of this definition of basic type is that the basic type of an order *i* (i>1) is the *powertype* of the basic type at the order immediately below (i-1), showing that the basic types are formed by the cascaded application of the powertype pattern. Furthermore, this cascade of basic types builds up from the constant “Individual”. This is reflected in the theorem T3, which is the result of applying D6 and A2 to D7. T3 simplifies the interpretation of D7 and can be read as “every basic type is either ‘Individual’ or the powertype of the basic type at the order immediately below”.

$$\forall b_i(\text{basictype}(b_i) \leftrightarrow$$
$$(b_i = \text{Individual}) \vee \exists b_{i-1}(\text{basictype}(b_{i-1}) \wedge \text{isPowertypeOf}(b_i, b_{i-1})))) \quad \text{(T3)}$$

Every ordered type that is not a basic type (e.g., a domain type) is an instance of one of the basic higher-order types (e.g., “1stOT”, “2ndOT”), and, at the same time proper specializes the basic type at the immediately lower level (respectively, “Individual” and “1stOT”). This treatment of type orders employed in MLT* meets requirement **R3** by defining a structure under which ordered entities can be interpreted. Figure 9 illustrates this pattern. Since “Person” applies to individuals, it is instance of “1stOT” and proper specializes “Individual”. The instances of “PersonTypeByGender” are specializations of “Person” (e.g. “Man” and “Woman”). Thus, “PersonTypeByGender” is instance of “2ndOT” and proper specializes “1stOT”.

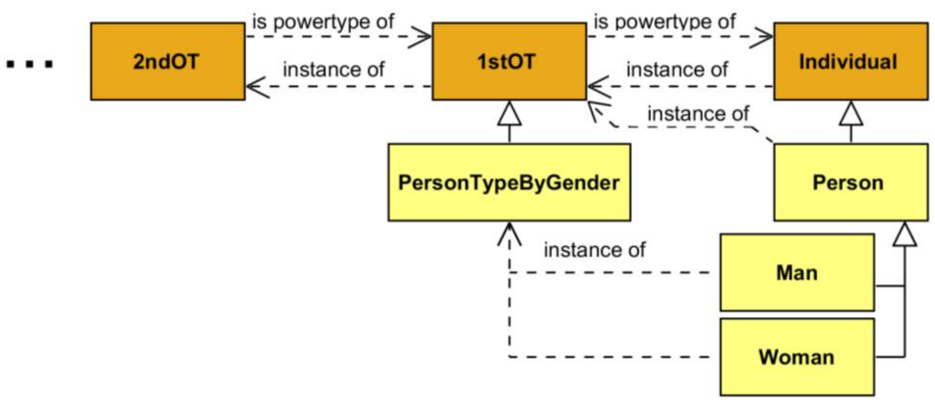


Figure 9 – Illustrating a basic pattern of MLT* and its intra-level structural relations.

Note that, the ellipsis in the left-hand side of the figure indicates that the theory admits an unbound number of higher-order basic types. Nevertheless, we have been careful not to necessitate the existence of such types in the theory. This means that the theory has finite models, and thus can be subject to analysis using a finite model checker/finder such as Alloy (Jackson, 2006), which we have employed for verification of all theorems discussed here.

Having defined the structure of basic types we can define ordered type as a type that specializes one of the basic types (D8). Conversely, we can defined *orderless types* as in D9.

$$\forall x(\mathrm{orderedtype}(x) \leftrightarrow \exists b(\mathrm{basictype}(b) \land \mathrm{specializes}(x,b))) \quad \text{(D8)}$$

$$\forall x(\mathrm{orderlesstype}(x) \leftrightarrow \mathrm{type}(x) \land \neg \mathrm{orderedtype}(x)) \quad \text{(D9)}$$

We can account now for a strictly stratified scheme. In this case, it would suffice to add an axiom stating that all types are ordered types, which would rule out types whose instances belong to different orders. The stratified scheme is thus a restriction of the more general theory we have, which admits *orderless types*.

Moreover, we can see that the theory can be further constrained to account for the two-level scheme as a particular case. For a two-level theory it would suffice to add to the strictly stratified scheme an axiom stating that there is a unique basic type (which would be "Individual").

3.3 Beyond Strictly Stratified Types

While a strictly stratified approach imposes a useful principle of organization for entities in multi-level models, it rules out types whose instances transcend this strict structure, i.e., types that have instances belonging to different levels or strata. For example, consider the type whose instances are all types admitted ("Type"). This type itself defies stratification into orders, since its instances are types at various different orders (e.g., "Individual", "Animal", "1stOT", "AnimalSpecies", "2ndOT", "Taxonomic Rank", etc.).

In order to capture the strictly stratified scheme while still guaranteeing the generality of the theory, MLT* distinguishes types into "OrderedType" (A3) and "OrderlessType" (A4). Instances of "OrderedType" are those types that fall neatly into a particular order. Instances of "OrderlessType" are those types whose instances do not belong to a single order. This constitutes a dichotomy, and together, "OrderedType" and "OrderlessType" form the notion of "Type" (A5), which classifies all possible types. In their turn "Type" and "Individual" (A2) together form the universal notion of "Entity" (A6), which classify all possible entities (types and individuals).

$$\forall t(t = \mathrm{OrderedType} \leftrightarrow \forall x(\mathrm{orderedtype}(x) \leftrightarrow \mathrm{iof(x,t)})) \quad \text{(A3)}$$

$$\forall \mathrm{t}(\mathrm{t} = \mathrm{OrderlessType} \leftrightarrow \forall \mathrm{x}(\mathrm{orderlesstype}(\mathrm{x}) \leftrightarrow \mathrm{iof}(\mathrm{x},\mathrm{t}))) \quad \text{(A4)}$$

$$\forall \mathrm{t}((\mathrm{t} = \mathrm{Type}) \leftrightarrow \forall \mathrm{x}(\mathrm{type}(\mathrm{x}) \leftrightarrow \mathrm{iof}(\mathrm{x},\mathrm{t}))) \quad \text{(A5)}$$

$$\forall \mathrm{t}(\mathrm{t} = \mathrm{Entity} \leftrightarrow \forall \mathrm{x}(\mathrm{iof}(\mathrm{x},\mathrm{t}))) \quad \text{(A6)}$$

Regarding to orderless types, their instances can be arranged in a multitude of ordering patterns: orderless types may classify entities of any order (e.g. “Entity” and “Type”, which classify both ordered and orderless types); or classify only entities of certain orders (e.g. “OrderedType”, which does not classify orderless types or individuals). The support for orderless types in MLT* allows it to meet requirement **R4**. The classification scheme formed by MLT* is presented in Figure 10.

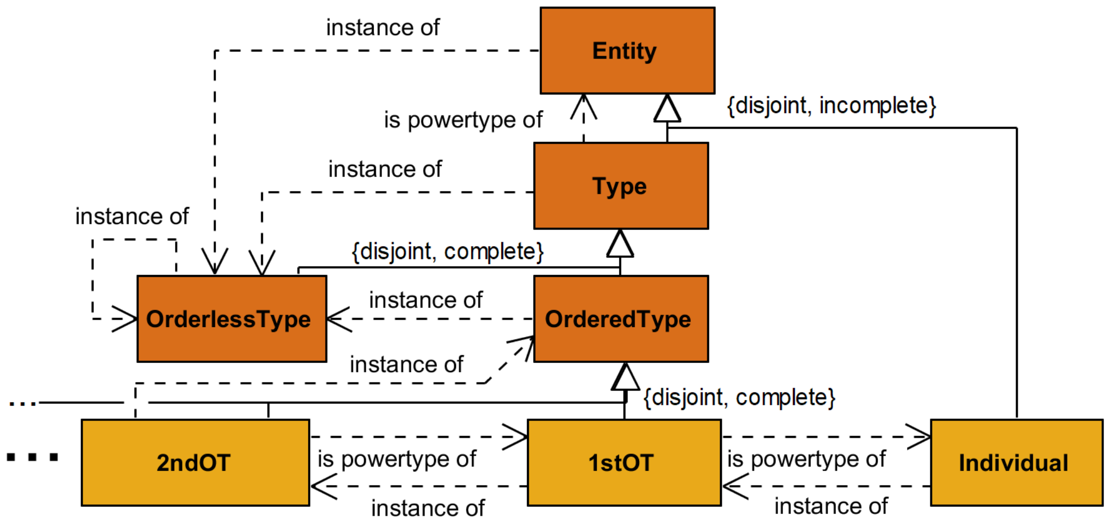


Figure 10 – MLT* classification scheme.

A number of interesting observations can be made about the top-layer of MLT*. First of all, MLT*, differently from MLT, is able to account for the types used in its definition. All entities admitted are instances of “Entity”, including all possible types and all possible individuals. All possible types are instances of “Type” and ultimately specializations of “Entity” (since their instances are entities). “Type” is thus the *powertype of* “Entity”. All elements added in MLT* are instances of “OrderlessType”, including (curiously) “OrderedType” (since its instances are types at different orders).

The instantiation relation has the following logical properties as a consequence of the definitions and axioms of the theory[8]: whenever instantiation involves solely ordered types, it is *irreflexive*, *antissymetric* and *antitransitive*, leading to a strict stratification of types. When instantiation involves any orderless types, none of these properties can be asserted, as there are situations in which it is *reflexive* (e.g., “Type” is instance of itself), *symmetric* (e.g., “Entity” is instance of “Type” and vice-versa) as well as *transitive* (e.g., “OrderedType” is instance of “Type” which is instance of “Entity” and “OrderedType” is also instance of “Entity”). Further, an orderless type is never an instance of an ordered type. These characteristics of instantiation can be used to rule out models that violate the theory.

[8] See https://github.com/nemo-ufes/mlt-ontology for the formalization of MLT* in Alloy, including assertions that have been verified corresponding to the theorems and properties we discuss here.

Table 3 summarizes the rules that concern which types of entities may be related through the structural relations presented so far along with the logical properties of these relations.

Table 3 – Summary of constraints on MLT* relations.

Relation (t → t')	Domain	Range	Constraint	Properties
specializes(t,t')	Orderless	Orderless	if *t* and *t'* are ordered types, they must be at the same type order	Reflexive, antissymetric, transitive
	Ordered	Orderless		
	Ordered	Ordered		
properSpecializes(t,t')	Orderless	Orderless		Irreflexive, antissymetric, transitive
	Ordered	Orderless		
	Ordered	Ordered		
isPowertypeOf(t,t')	Orderless	Orderless	*t* cannot be a first-order type if *t* and *t'* are ordered types, *t* must be at a type order immediately above the order of *t'*	Irreflexive, antissymetric, antitransitive
	Ordered	Ordered		

The notion of "Orderless Type" is useful not only for the domain-independent entities forming MLT*, but also for general notions in specific subject domains. Consider, for example, the domain of social entities in which a "Social Entity" is defined as an entity that is created by a social normative act. Instances of "Social Entity" include specific states of Brazil (individuals) such as "Rio de Janeiro" and "Espírito Santo", but also the first-order type "State" of which "Rio de Janeiro" and "Espírito Santo" are instances. As "SocialEntity" has instances at different orders (types and individuals), it is an instance of "OrderlessType", as shown in Figure 11. The example also highlights that MLT* allows entities to have multiple instantiation relations. "RioDeJaneiro" and "EspíritoSanto" are both instances of "SocialEntity" and "State". Moreover, multiple specializations are also allowed in MLT*. In this sense, MLT* differs from a number of approaches in literature which limit these structural relations to a single class (see (Lara, Guerra and Cuadrado, 2014)).

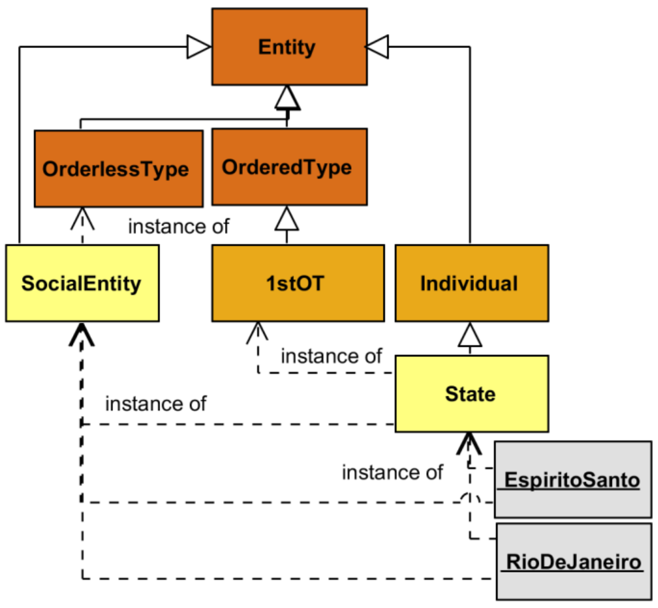


Figure 11 – Example of orderless type in domain model

The same mechanism that allows us to model *bona fide* self-instantiating types such as “Entity” and “Type”, would permit a modeler to introduce paradoxical types such as the type of all types that are not self-instantiated (the so-called Russellian property, due to Russell (Irvine and Deutsch, 2016)). This type is paradoxical since it is both an instance and not an instance of itself. Note that this possibility does not threaten the overall consistency of the theory. This is because we do not assume in MLT* that there are types corresponding to any expressible unifying condition (i.e., we do not assume that given an arbitrary logical condition Y, we can define the type with extension [x | Y(x)]). Types here, instead, are explicitly recognized entities describing intentionally identified properties shared by their instances. Lacking the ability to prove or introduce the existence of types in this sense, we are under no threat of such paradoxes (Menzel, 2011).

3.4 Structural Relations for Multi-Level Modeling

So far, the only *cross-level structural relations* we have considered is Cardelli’s powertype relation. Another definition of *powertype* that has had great influence in the literature was proposed by (Odell, 1994). In order to satisfy **R5**, and account for the variations of the powertype pattern in the literature, MLT* defines the categorization cross-level relation based on Odell’s notion powertype.

As defined in D10, a type t_1 *categorizes* a type t_2 iff all instances of t_1 are proper specializations of t_2. Note that, differently from the *is powertype of* relation (due to Cardelli), t_2 is not an instance of t_1, and further not all possible specializations of t_2 are instances of t_1. For instance, “EmployeeType” (with instances “Manager” and “Researcher”) *categorizes* “Person”, but is not the *powertype* of “Person”, since there are specializations of “Person” that are not instances of “EmployeeType” (“Child” and “Adult” for example).

$$\forall t_1, t_2(\text{categorizes}(t_1, t_2) \leftrightarrow$$
$$(\neg\text{iof}(t_1, \text{Individual}) \land \forall t_3(\text{iof}(t_3, t_1) \rightarrow \text{properSpecializes}(t_3, t_2)))) \quad \text{(D10)}$$

MLT* (borrowing from MLT) also defines some variations of the categorization relation. A type t_1 *completely categorizes* a type t_2 iff every instance of t_2 is instance of at least one instance of t_1 (D11). Moreover, a type t_1 *disjointly categorizes* a type t_2 iff every instance of t_2 is instance of at most one instance of t_1 (D12). Further, t_1 *partitions* t_2 iff every instance of t_2 is instance of exactly one instance of t_1 (D13).

$$\forall t_1, t_2(\text{completelyCategorizes}(t_1, t_2) \leftrightarrow$$
$$(\text{categorizes}(t_1, t_2) \land \forall e(\text{iof}(e, t_2) \rightarrow \exists t_3((\text{iof}(e, t_3) \land \text{iof}(t_3, t_1)))))) \quad \text{(D11)}$$
$$\forall t_1, t_2(\text{disjointlyCategorizes}(t_1, t_2) \leftrightarrow$$
$$(\text{categorizes}(t_1, t_2) \land \forall e, t_3, t_4((\text{iof}(t_3, t_1) \land \text{iof}(t_4, t_1) \land \text{iof}(e, t_3) \land \text{iof}(e, t_4)) \rightarrow t_3 = t_4))) \quad \text{(D12)}$$

$$\forall t_1, t_2(\text{partitions}(t_1, t_2) \leftrightarrow$$
$$(\text{completelyCategorizes}(t_1, t_2) \wedge \text{disjointlyCategorizes}(t_1, t_2))) \quad \text{(D13)}$$

In order to illustrate the usage of these relations, Figure 12 shows some examples. “PersonTypeByGender” *partitions* “Person” into “Man” and “Woman”, and thus each instance of “Person” is either a “Man” or a “Woman” and not both. “EmployeeType” incompletely *categorizes* “Person”, and thus there are persons that are not instances of “Manager”, “Researcher” (or any other possible instance of “EmployeeType”). This kind of constraint is usually represented in UML through a generalization set, however the semantics differs from the variations of categorization presented here (see (Carvalho and Almeida, 2016) for a detailed comparison).

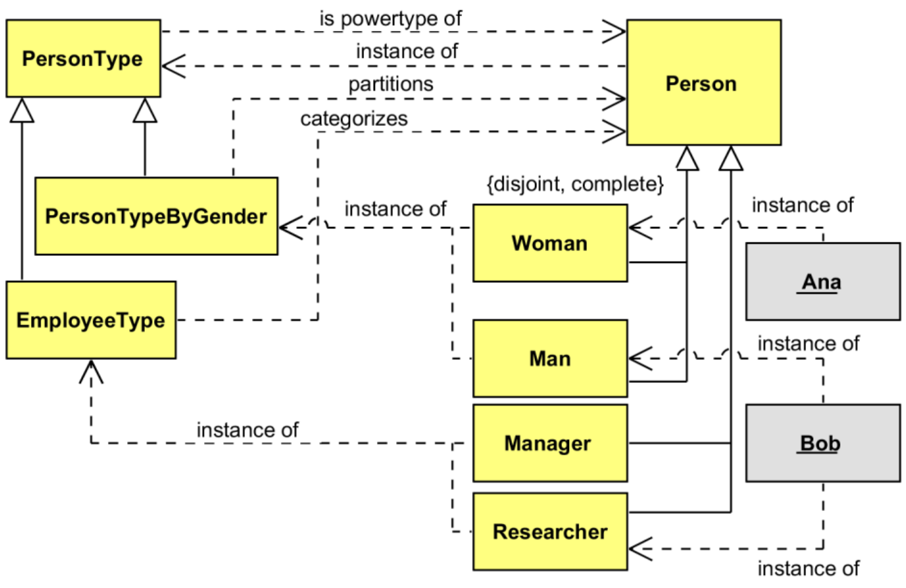


Figure 12 – Example of categorization and partitions relations.

Finally, MLT and MLT* also account for another kind of *intra-level structural relation*, in addition to the specialization relation. The subordination relation allows the specification of hierarchies of specialization between instances of two types. More precisely, *t_1 is subordinated to t_2* iff every instance of t_1 proper specializes some instance of t_2. Since subordination implies proper specializations between the instances of the involved types at one order lower, subordination can only involves orderless types or higher-order types of equal order.

$$\forall t_1, t_2(\text{isSubordinate}(t_1, t_2) \leftrightarrow (\neg \text{iof}(t_1, \text{Individual}) \wedge \forall \text{t}_3(\text{iof}(\text{t}_3, \text{t}_1) \rightarrow$$
$$\exists \text{t}_4((\text{iof}(\text{t}_4, \text{t}_2) \wedge \text{properSpecializes}(\text{t}_3, \text{t}_4)))))) \quad \text{(D14)}$$

In order to illustrate this relation, take for instance the biological domain example presented earlier, with the types “Species” and “Breed”. In this example, “Species” and “Breed” are second-order types that have as instances “Dog” and “Collie” respectively. As every instance of “Breed” classifies instances of some instance of “Species”, “Breed” is subordinated to

"Species" (see Figure 13). As shown in (Carvalho and Almeida, 2016), subordination is key to the representation of the relations between the various biological taxonomy ranks.

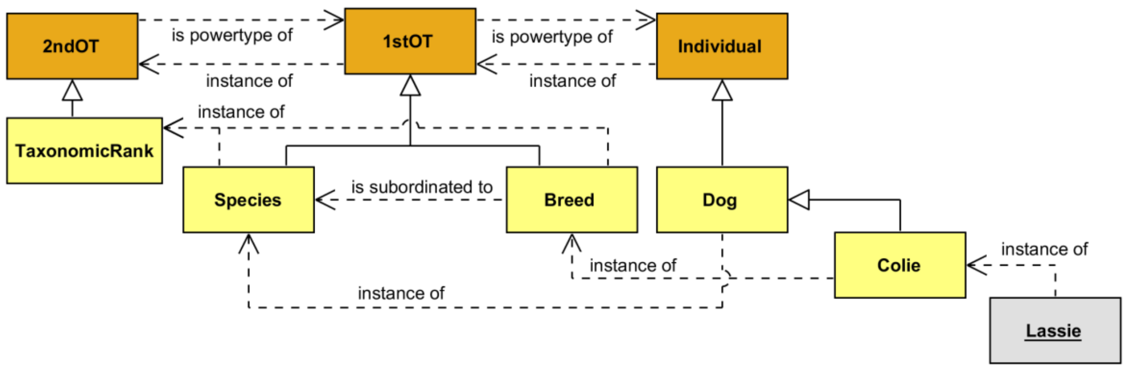


Figure 13 – Subordination example between "Species" and "Breed".

Rules concerning the types of entities that may be related through the variations of categorization and subordination in addition to the logical properties of these relations are summarized in Table 4.

Table 4 – Summary of constraints on MLT* categorization and subordination relations.

Relation (t → t')	**Domain**	**Range**	**Constraint**	**Properties**
isSubordinatedTo(t,t')	Orderless	Orderless	*t* and *t'* cannot be first-order types	Irreflexive, antissymetric, transitive
	Ordered	Orderless	if *t* and *t'* are ordered types, they must be at the same type order	
	Ordered	Ordered		
categorizes(t,t') *disjointlyCategorizes(t,t')*	Orderless	Orderless	*t* cannot be a first-order type	Irreflexive, antissymetric, nontransitive
	Ordered	Orderless		
	Ordered	Ordered	if *t* and *t'* are ordered types, *t* must be at a type order immediately above the order of *t'*	
completelyCategorizes(t,t') *partitions(t,t')*	Orderless	Orderless		Irreflexive, antissymetric, antitransitive
	Ordered	Ordered		

3.5 Conclusion

In this chapter, we have presented MLT* as the theoretical basis for the development of our multi-level modeling language. This theoretical basis builds upon MLT (Carvalho and Almeida, 2016) to allow a more general interpretation of the instantiation relation among entities of a conceptual domain. Such generalization of the original theory allows the discussion of conceptual entities that go beyond strictly stratified schemes. Besides the definition of the notion of orderless type, MLT* also differ from MLT by providing a general definition of basic types and a well-founded definition of *type*. In addition, all the theorems and axioms of MLT are also adapted and expanded for the formalization of MLT*.

MLT* shows that there is no dilemma in supporting orderless types in combination with stratified schemes. This presents an opportunity for the extension of other approaches which focus on the organization of entities into levels, such as Melanee (Atkinson and Gerbig, 2012) and MetaDepth (de Lara and Guerra, 2010). For example, Melanee and MetaDepth could work out a mechanism to allow some kind of selective stratification, beyond what is currently supported with the so-called star potency, in order to fully enable the representation of orderless types.

Further, MLT* also shows that powertype-based and clabject-based approaches can be harmonized. By giving support to both mechanisms, the approach leaves to the user the choice of representing the basetype according to what is most suitable for the subject domain. Giving support to variations of the powertype is also an extension opportunity for languages such as Deeptelos (Jeusfeld and Neumayr, 2016), since the relevance of these variations presents representation benefits in many domains.

Regarding the requirements for conceptual modeling, Table 5 summarizes the strategy for employing MLT* to fulfill requirements R1 to R5. Requirements R6 to R8 will be addressed later in the Chapter 4 with the treatment of features.

Table 5 – Summary of multi-level modeling requirements and fulfillment strategies.

Requirement	Strategy
R1 – represents entities of multiple classification levels	Definition of an *instance of* relation applicable to any kind of entity (either types or individuals).
R2 – represents arbitrary number of classification levels	Unrestricted application of the *instance of* relation among entities allowing instantiation chains of any size.
R3 – defines principles for organization of models into levels	Definition of a conceptual layer for interpreting ordered entities.
R4 – admits types that defy a stratified classification scheme	Unrestricted application of the *instance of* relation associated to a conceptual layer for interpreting orderless entities.
R5 – accounts for rules to govern instantiation of related types	Definition of a set of structural cross-level relations based on major strategies in the literature for modeling multi-level domains.

Chapter 4. ML2: The Multi-Level Modeling Language

Having defined a theory for multi-level modeling, in this chapter in employ MLT* on the development of a multi-level language. The Multi-Level Modeling Language (ML2) is a textual language that reflects the concepts and rules of MLT*. In addition, the rules that constitute MLT* (definitions and theorems) guide the language's *semantically-motivated syntactic constraints*. Since it is based on a formal theory, the language constructs have a clear semantics, which improves model quality. With a focus on expressivity and model readability, ML2's syntax is largely inspired in major OO programming languages. This chapter is divided in the following sections: sections 4.1 to 4.3 presents the ML2 language, considering its abstract and concrete syntaxes; Section 0 presents a list of syntactic rules for the language that are reflect rules and theorems from the MLT* theory. Throughout the chapter, we discuss how the requirements are satisfied by the language.

4.1 Modeling Multi-Level Entities

4.1.1 Core Constructs

The linguistic constructs of ML2 aim at reflecting the conceptual backbone of MLT* delivering to the user language features that represent types of entities and relations defined by the theory. As shown in Figure 14, the portion of the language's metamodel (an Ecore metamodel) regarding entities (individuals and types) reflects the basic scheme of MLT*. Aside from minor terminological differences (with *Class* replacing *Type* for consistency with EMF terminology, and *EntityDeclaration* replacing *Entity*), there are corresponding constructs for all concepts presented earlier in Figure 10. In the metamodel, only the classes in gray can be instantiated through language constructs.

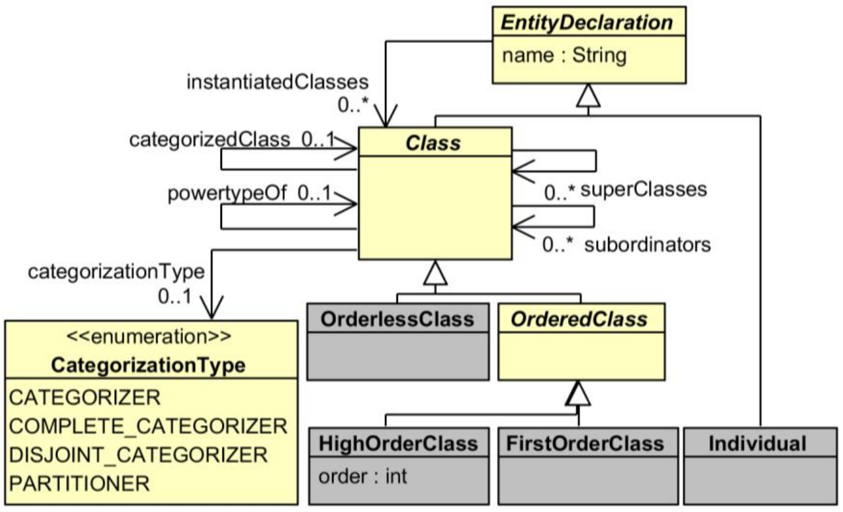


Figure 14 - Entities and classes in ML2

"HighOrderClass" captures ordered classes representing theirs orders through the "order" attribute. Rather than requiring explicit references to some *basic type* or relying on the size of instantiation chains for inference of the entity's order, this allows an intuitive declaration of strictly stratified types that strongly resembles potency-based approaches.

Besides providing support for entity in ordered structures, ML2 also provides MLT*'s structural relations as language constructs, and by doing so, the language is able to meet requirements R1 to R5 discussed in Chapter 2. ML2 differs from many modeling languages (such as Melanee (Atkinson and Gerbig, 2012) and MetaDepth (de Lara and Guerra, 2010)) by allowing declaration of multiple instantiation, (proper) specialization and subordination relations. The instantiation relation can be declared for any entity, while the rest of the structural relations are always declared between classes. In the case of categorizations, an additional enumeration identifies the type of relation held towards the categorized class. The language's *semantically-motivated syntactic constraints* directly reflect those presented earlier in Table 3 and Table 4. These constraints are verified on the model through a validation mechanism that is part of ML2's editor, a topic that will be discussed later.

Many of the relations in MLT* can be inferred from other relations. For this reason, ML2 is ready to deal with a minimal usage of structural relations, which improves the readability of models by keeping the declarations as simple as possible. For instance, if an individual "John" instantiates both first-order classes "Man" and "Person", only the more specific instantiation needs to be declared. As long as "Man" holds a specialization towards "Person", the instantiation from "John" to "Person" is inferred through its instantiation towards "Man". Nevertheless, both instantiations can be declared without any harm to the model's semantics, what is even encouraged if the modeler wants to help a human reader to interpret the instantiations of "John" without a prior knowledge of the specialization hierarchy for "Man".

ML2 uses a textual syntax largely inspired in traditional OO languages, and applies a collection of keywords aiming at enhancing the readability of its models. The statements for entity declaration follow a common pattern, varying the available structural relations for each type of entity. Figure 15 shows a fragment of ML2's syntax for entity declaration in a BNF-like syntax. The declaration of an orderless class, for instance, starts with the keywords "orderless class", followed by the entity's name, and its structural relations, closing with a semi-colon. Throughout the text, we present the syntax in this style, where '?', '*' and '|' represent, respectively, optional statements, repeatable statements and alternatives. Terms in bold represent terminal symbols and terms in italics represent cross-references (i.e., identifiers that refer to another model element).

```
Entity := Class | Individual ;
Class := (FirstOrderClass | HighOrderClass | OrderlessClass)
                ({ (Feature | FeatureAssignment)* })?


FirstOrderClass := class NAME MLTRelations*
HighOrderClass := order NUMBER class NAME MLTRelations* ;
OrderlessClass := orderless class NAME MLTRelations* ;
Individual := individual NAME Instantiation ({ (Feature | FeatureAssignment)* })?


MLTRelations := Instantiation | Specialization | Subordination | Powertyping |
                Categorization


Instantiation := : Class (, Class)*
Specialization := specializes Class (, Class)*
Subordination := subordinatedTo Class (, Class)*
Powertyping := isPowertypeOf Class
Categorization := CategorizationType Class


CategorizationType := categorizes | completeCategorizes | disjointCategorizes | partitions
```

Figure 15 – Entity declaration syntax

Figure 16 revisits the examples from Chapter 3 using ML2. Note that, a namespace mechanism is supported with modules, which are fragments of models that contain ML2 model elements. Individuals are the only entities that require some instantiation declaration, possibly instantiating multiple types (e.g., "Eva" instantiates "Person", "Manager" and "Adult", and "Bob" instantiates "Person" and "Child").

```
module example.model {
        orderless class SocialEntity;

        order 2 class PersonPowertype isPowertypeOf Person;
        order 2 class PersonTypeByAge specializes PersonPowertype partitions Person;
        order 2 class EmployeeType specializes PersonPowertype categorizes Person;

        class Person : PersonPowertype;
        class Manager : EmployeeType specializes Person;
        class Researcher : EmployeeType specializes Person;
        class Child : PersonTypeByAge specializes Person;
        class Adult : PersonTypeByAge specializes Person;

        individual Eva :Person, Manager, Adult;
        individual Bob :Person, Child;

        class State : SocialEntity;
        individual EspiritoSanto : State, SocialEntity;
        individual RioDeJaneiro : State, SocialEntity;
}
```

Figure 16 – Examples of entity declarations in ML2

4.1.2 Generalization Sets

Considering the capability of aggregating specializations defined from a common criterion, ML2 borrows from UML (OMG, 2011) the concept of *generalization set* (see the metamodel's fragment in Figure 17). A generalization set links a super class (called *general*) to a set of specializations of it (the *specifics*). Generalization set can be *complete*, in cases where instances of the general class must instantiate at least one of the specifics classes, and *disjoint*, in cases where instances of the general class can instantiate at most one of the specifics classes. In addition, a generalization set may complement representation of categorization relations when the specific classes of the set are instances of a single categorizer of the general class.

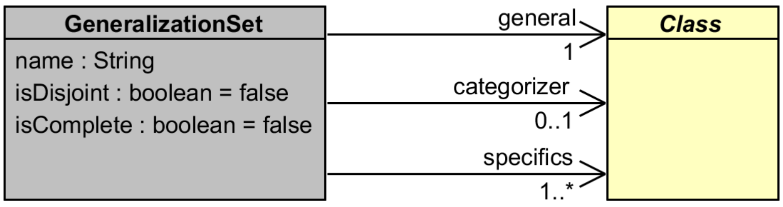


Figure 17 – Generalization sets in ML2.

The syntax for declaration of generalization set in ML2 is shown on Figure 18.

```
GeneralizationSet := disjoint? complete? genset NAME?
                         general Class
                         (categorizer Class)?
                         specifics Class (, Class)*;
```

Figure 18 – Generalization set syntax.

In order to illustrate the fully of application this language construct, Figure 19 shows an expanded version of the example previously seen in Figure 16. In this case, we add instances of “PersonTypeByAge” to include all possible instances of it. Notice that, while the partitions relation from “PersonTypeByAge” towards “Person” defines that every instance of the latter must instantiate an instance of the former, the completeness constraint of the generalization set states the every instance of “Person” instantiates one of the classes within that set. These are two related but not equivalent rules and the generalization set makes it clear that a person “Person” must instantiate an instance “PersonTypeByAge” declared there, and not in a further module.

```
 order 2 class PersonTypeByAge specializes PersonPowertype partitions Person;

class Person : PersonPowertype;
class Child : PersonTypeByAge specializes Person;
class Teenager : PersonTypeByAge specializes Person;
class Adult : PersonTypeByAge specializes Person;
class Elder : PersonTypeByAge specializes Person;

disjoint complete genset person_by_age
        general Person
        categorizer PersonTypeByAge
        specifics Child, Teenager, Adult, Elder;
```

Figure 19 – Examples of generalization set in ML2

The combination of disjointness and completeness constraints from generalization sets and categorization relations was the subject of investigation of Carvalho, Almeida and Guizzardi (2016), who analyzed this combination in an MLT extension for the UML language (OMG, 2011). The set of possible combinations of these constraints is summarized in Table 6 having the following interpretation:

- **Enumerated:** in the enumerated combination, the generalization set contains all possible instances of the categorizer type. It only occurs in cases where the categorizer partitions the general class of a disjoint and complete generalization set;
- **Not Enumerated:** in the “not enumerated” combination, some possible instance of the categorizer is not included in the generalization set. It occurs in cases where the generalization set is not complete, or the categorizer does not disjointly

categorizes the general class, allowing overlapping between instances of the categorizer that are present in the generalization set with those that are not;

- **Invalid:** invalid combinations occur when the constraints from the generalization set are conflicting with those from the categorization. It occurs when the generalization set allows overlapping between instances of a disjoint categorization, and when the instantiation of a non-complete categorization (i.e., simple or disjoint categorization) is enforced by a complete generalization set;
- **Silent:** finally, silent combinations are valid but do not allow the inference of an enumerated, or "not enumerated", set of instances of the categorizer.

Table 6 – Analyzing the combination of categorization and generalization sets (adapted from (Carvalho, Almeida and Guizzardi, 2016)).

Categorization Relation	**Generalization Set Constraints**			
	Disjoint		***Overlapping***	
	Complete	***Incomplete***	***Complete***	***Incomplete***
Partitons	Enumerated	Not Enumerated	Invalid	Invalid
Disjoint Categorization	Invalid	Silent	Invalid	Invalid
Complete Categorization	Not Enumerated	Not Enumerated	Silent	Not Enumerated
Categorization	Invalid	Not Enumerated	Invalid	Silent

4.2 Features and Assignments

Classes, in conceptual modeling, are classifiers applicable to entities that share a common set of features. Alternatively, entities are instances of the classes that aggregate the features that describe them. Since (Chen, 1976), it is common to represent features of entities in a conceptual model through *attributes* and *relationships*. The attributes and relationships capture the characteristics of entities in general terms without applying concrete values to each instance. This allows for entities that share a certain characteristic (e.g. weight or height) to have different concrete values for it (e.g. "John" weighs 70kg while "Bob" weighs 80kg). In general, modeling solutions that adopt two-level schemes allow the specification of attributes and relationships at the *type-level*, leaving the assignment of values of features for the specification of instances when necessary. However, multi-level conceptual domains require the capacity of representing both instances and classifiers together, which leads to the necessity of representing both features and values for any classified entity in the model.

This is supported in ML2 with the mechanisms in the metamodel fragment shown in Figure 20. Note that FeatureAssignment is defined for Entities in general (including classes and individuals), while Features can be specified for any class (regardless of order). Since any instances may contain assignments for instantiated features, ML2 satisfies requirement **R6**.

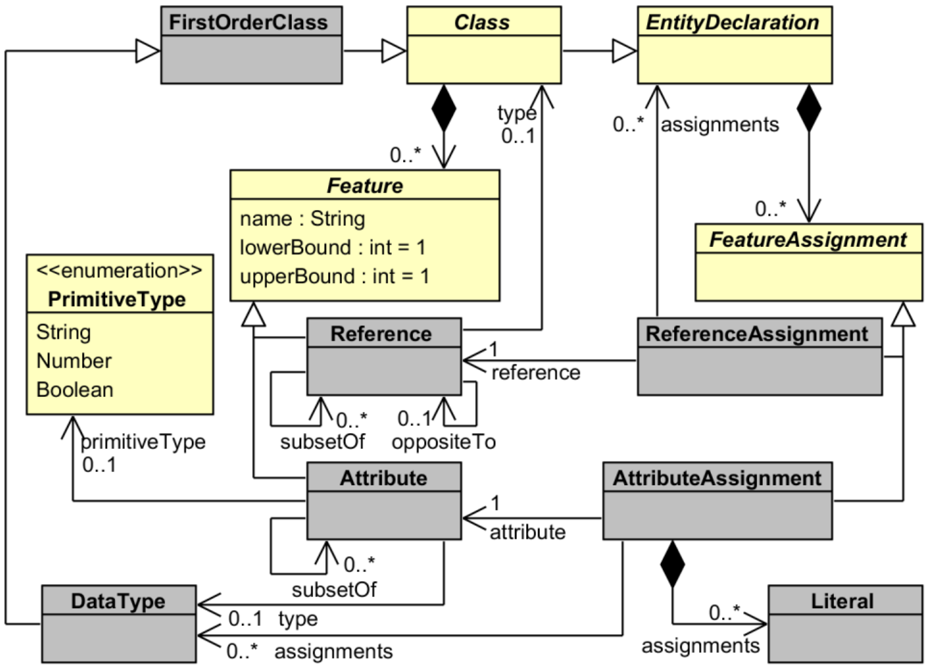


Figure 20 – Features and assignments in ML2

Typically, features are mutable elements and their assignments may change in time. However, temporal aspects are not explicitly dealt with in ML2 models, therefore, these models must interpreted as the representation of the state-of-affairs of a domain in a particular point in time (capturing, thus, a “snapshot” view of world).

ML2 distinguishes features into references and attributes (not unlike Ecore and OWL, for example). References can relate instances of any two classes (from its containing classes towards the reference type), besides being able to subset references from specialized classes as well as being opposite to some reference of inverse direction. The subsetting mechanism allows features of a specialized class to refine inherited features by determining more specific types and narrower cardinalities. Opposite references is a mechanism for dealing with simple associations in ML2. Associations with features and associations with an arity higher than two can be modeled through reification, using the ontologically well-founded notion of relators as discussed in (Guarino and Guizzardi, 2015; Carvalho and Almeida, 2016). The same approach lends itself to considering high-order types for relators when necessary, as shown in (Carvalho and Almeida, 2015).

An attribute, differently from a reference, can only have a *data type* as its type, be it a primitive type or a user defined data type. Data types are first-order classes that have as instances particular values, for example the data type *String*, which has as instances any well-formed sequence of characters. The set of primitive types in ML2 (*String*, *Number* and *Boolean*) covers a minimal set of data types for conceptual modeling and was inspired in JSON’s specification (ECMA, 2013). Figure 21 presents the syntax for feature declaration in ML2.

Literals are employed to represent assignments of attributes based on primitive types, being a number, a string or a declaration of “true” or “false”.

```
Feature := Reference | Attribute
Reference := ref NAME : Multiplicity? Class Subsets? IsOpposite?
Attribute := att NAME : Multiplicity? Datatype Subsets?

Multiplicity := [ CARD .. CARD ]
Subsets := subsets Feature (, Feature)*
IsOpposite := isOppositeTo Reference

DataType := datatype NAME MLTRelations* ({ (Feature | FeatureAssignment)* })? ;


FeatureAssignment := ReferenceAssignment | AttributeAssignment
ReferenceAssignment := ref Feature = Entity | { Entity (, Entity)* }
AttributeAssignment := att? Feature = Literal | { Literal (,Literal)* }


Literal := STRING | NUMBER | BOOLEAN | Entity
```

Figure 21 – Feature declaration syntax

Figure 22 presents an example of usage of features in an ML2 model. This model expands the one presented in Figure 16 and includes the reference *is designed by*, a case of references between entities of different classification levels (satisfying requirement **R8**) presented earlier in Chapter 2. Note that ML2 does not require exhaustive feature assignment (see instances of “Person”) in order to allow for partial (incomplete) models. Nevertheless, whenever there is a feature assignment, cardinality constraints as well as feature type must be respected, and corresponding syntactic constraints are foreseen. For example, “Eva” is an instance of “Person” and, thus, has the feature “isOffspringOf”, which has no assignment in this case. On the other hand, “Bob” has an assignment for the feature “isOffspringOf”, which must obey the cardinality and type constraints of the feature (i.e., “isOffspringOf” exactly two instances of “Person”, “Jony” and “Eva”).

```
order 2 class CellphoneModel categorizes Cellphone {
        ref isDesignedBy : Person
};
class Cellphone {
        ref owner : Person
        screenSize : Number
        color : Color
};
class IPhone5 : CellphoneModel specializes Cellphone{
        ref isDesignedBy = Jony
};


class Person : PersonPowertype {
        ref isOffspringOf : [2..2] Person isOppositeTo isParentOf
        ref isParentOf : [0..*] Person isOppositeTo isOffspringOf
        age : Number
        alias : [0..*] String
        name : String subsets alias
};

datatype Color { red:Number green:Number blue:Number };
individual Black : Color { red=0 green=0 blue=0 };


individual JonysIPhone : IPhone5{
        ref owner = Jony
        screenSize = 4
        color = Black
};
individual Jony : Person, Adult {
        ref isParentOf=Bob
        alias = {"Jonathan", "Big J", "Jony"}
        name = "Jonathan"
};
individual Eva : Person, Manager, Adult { ref isParentOf=Bob };
individual Bob : Person, Child{ ref isOffspringOf={Jony,Eva} };
```

Figure 22 – Examples of features in ML2

4.3 Regularity Features

In a multi-level conceptualization, a particular phenomenon arises in classes that classify other classes: related features. To illustrate this, let us further develop the example presented in Figure 22. In the cellphone domain, "screenSize" is a feature of individual instances of "Cellphone", since it refers to a physical characteristic of the individual. Due to the nature of cellphone manufacturing, it is usual that all instances of a cellphone model (e.g., "IPhone5") share the same value of "screenSize". This domain characteristic allows the addition of an "instancesScreenSize" feature for "CellphoneModel", which represents the value of "screenSize" for every instance of a particular cellphone model. In this case, "screenSize" and "instancesScreenSize" are related features, more precisely, instances of "Cellphone" have their values of "screenSize" determined by the "instancesScreenSize" feature when they instantiate a particular cellphone model. For example, "IPhone5" is an instance of "CellphoneModel" with "instancesScreenSize" of 4", therefore "IPhone5" must have a "screenSize" of 4" as well.

The original proposal of MLT defines this special kind of feature called *regularity feature* (see (Guizzardi *et al.*, 2015; Carvalho and Almeida, 2016)). By definition, a regularity feature (also valid for MLT*) has the characteristic of constraining features at a lower level. Figure 23 presents an additional fragment of ML2's metamodel containing the mechanisms of the language for handling this kind of feature. A feature is considered a regularity feature whenever a regularity type is defined and the regulated feature is identified. A regularity feature may only exist in high-order or orderless types, since it constrains another type feature at a lower level. Moreover, this high-order or orderless type must categorize the type containing the regulated feature in order to ensure that every instance of the former inherits the regulated feature of the later. This mechanism of regularity features present in ML2 meets our last requirement, **R7**.

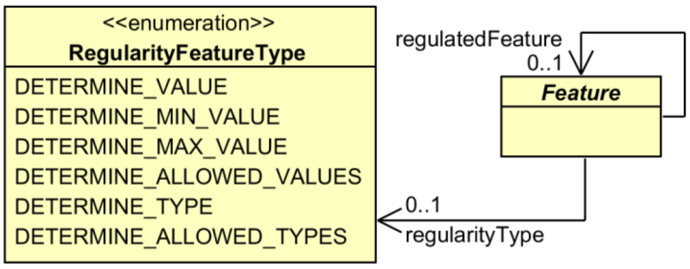


Figure 23 - Metamodeling of regularity features

ML2 foresees six types of regularity features. In the case above, values of "instancesScreenSize" determines the exact value of "screenSize". However, a regularity feature can also determine maximum or minimum values for a number feature (e.g., to model the maximum storage capacity of a cellphone model) and to determine the set of allowed values for a feature (e.g., to model that a phone model has either 16 or 32GB of internal storage capacity). Additionally, a regularity feature can further constrain the type of assignment for a feature, by either determining its type(s) or determining a set of allowed types. Figure 24 presents the ML2 syntax for declaring regularity features, in effect, redefining the "Feature" rule. The specification of the regularity type is optional in order to support the declaration of regularity features whose relation to the regulated feature is not covered by any of the foreseen types.

```
Feature := Reference | Attribute | RegularityReference | RegularityAttribute
RegularityReference := regularity Reference RegularityType? Feature
RegularityAttribute := regularity Attribute RegularityType? Feature
RegularityType := determinesValue | determinesMinValue | determinesMaxValue
                | determinesAllowedValues | determinesType | determinesAllowedTypes
```

Figure 24 – Regularity features syntax

In Figure 25, we present an example in which the regularity reference "compatibleProcessorModel" of "CellphoneModel" determines the type of "installedProcessor" for instances of "Cellphone". Since "IPhone5" assigns "A6" to "compatibleProcessorModel", instances of "IPhone5" can only have processors that are instances of "A6". This is the case of "JonysIPhone", with "Processor01" installed on it. Note that, if the instantiated regularity feature adds enough information about the domain, specifying values on the affected entities becomes unnecessary. For example, there is no need to assign the value of "screenSize" for "JonysIPhone" because all instances of "IPhone5" have this feature value determined by "instancesScreenSize" ("instancesScreenSize = 4"). When assignments of regulated features are present, they must respect the assignment of the associated regularity feature. This is part of the syntactic constraints of the language, and are thus verified by the editor.

```
order 2 class CellphoneModel categorizes Cellphone {
        regularity instancesScreenSize : Number determinesValue screenSize
        regularity ref compatibleProcessorModel : ProcessorModel
                determinesType installedProcessor
};
class Cellphone {
        screenSize:Number
        ref installedProcessor : Processor
};
class IPhone5 : CellphoneModel specializes Cellphone {
        instancesScreenSize = 4
        ref compatibleProcessorModel = A6
};

order 2 class ProcessorModel categorizes Processor;
class Processor;
class A6 : ProcessorModel specializes Processor;

individual Processor01 : A6;
individual JonysIPhone : IPhone5 {
        screenSize = 4
        ref installedProcessor = Processor01
};
```

Figure 25 – An example of regularity features

Table 7 summarizes how ML2 deals with the remaining requirements for a multi-level modeling language. Moreover, Figure 26 presents the complete metamodel of the ML2 language.

Table 7 – Summary of multi-level modeling requirements and fulfillment strategies.

Requirement	Strategy
R6 – represents features and feature assignments	Definition of appropriate language constructs to capture class features and entity assignments.
R7 – relates features of entities in different levels	Usage of *regularity features* to specify features that have effects over features of instantiating classes.
R8 - domain relations between entities in various levels	Unrestricted application usage of references according to their orders.

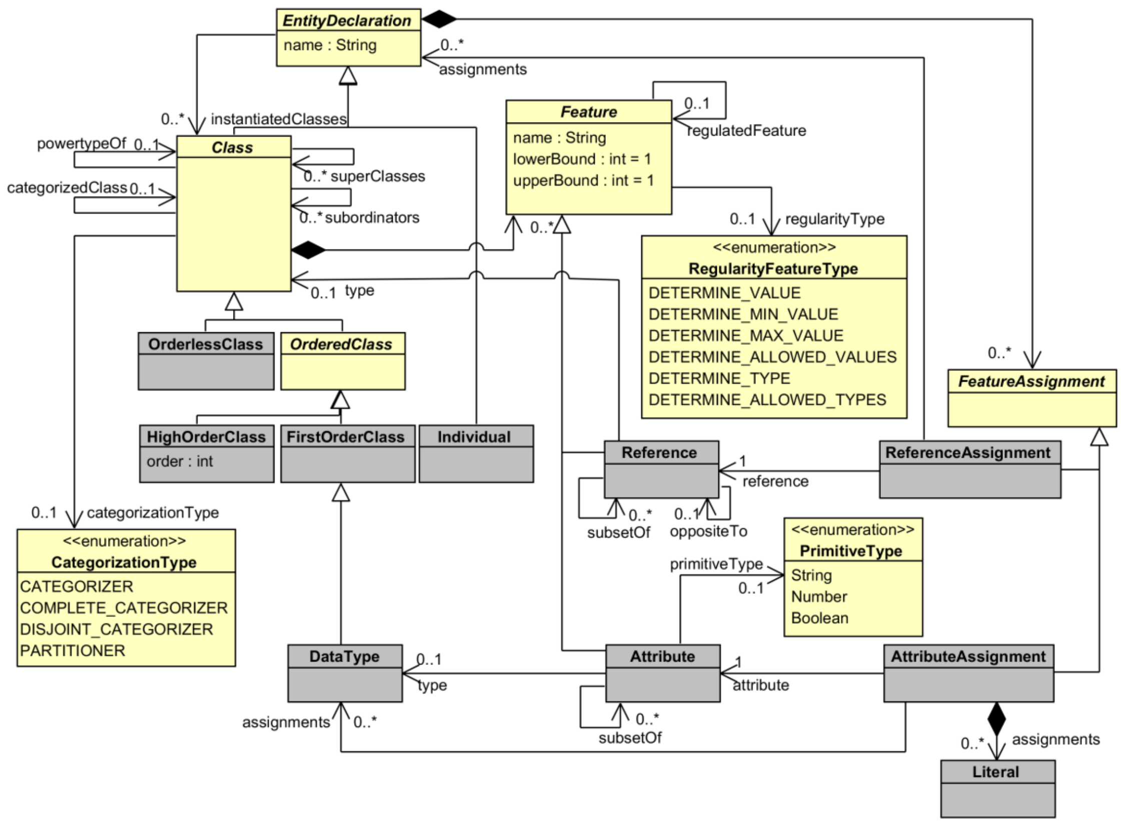


Figure 26 – Complete ML2 metamodel.

4.4 Syntactic Constraints

In addition to their impact on the metamodel and grammar of ML2, the definitions and theorems of MLT* inspire the specification of *semantically-motivated syntactic constraints*. These constraints rules out a number of possible models inconsistent according to the reference theory, for example, in terms of relation cycles, symmetry or transitivity. A summary of these rules, along with the types of entities they apply to, is presented in Table 8.

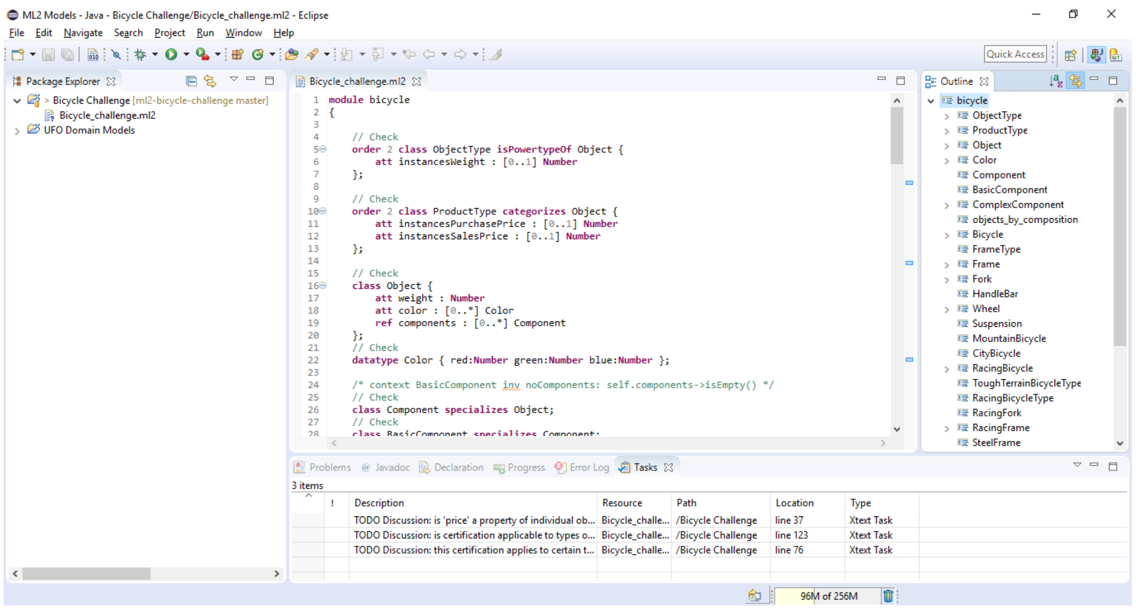
Table 8 – Summary of validation rules of ML2.

Type	Syntactic Rules
EntityDeclaration	An entity cannot instantiate disjoint classes.
Class	Specializations can only occurs between entities of same order or orderless classes (cf. Table 3).
Class	Classes cannot be in a specialization cycle (cf. Table 3).
Class	Orderless classes can only categorize other orderless classes (cf. Table 4).
Class	Ordered classes can only categorize classes of the order immediately below or orderless class (except in the cases of complete categorization and partitioning) (cf. Table 4).
Class	Orderless classes can only be powertype of others orderless classes (cf. Table 3).
Class	Ordered classes can only be powertype of classes in the order immediately below (cf. Table 3).
Class	Orderless classes can only be subordinated to others orderless classes (cf. Table 4).
Class	Ordered classes can only be subordinated to orderless classes and classes of the same order (cf. Table 4).
Class	Classes cannot be in subordination cycles (cf. Table 4).
Class	A class that instantiates a powertype must specialize its basetype.
Class	An instance of a subordinated class must specializes some instance of the related subordinator class.
Class	A class cannot specialize disjoint classes.
Class	A class must assign values to the regularity features it instatiates.
HighOrderClass	High-order classes must have an order higher than 1.
DataType	Datatypes cannot contain references.
GeneralizationSet	The categorizer class must categorize the general class.
GeneralizationSet	The specific classes of a generalization must be direct instances of the categorizer type when it is present.
Feature	Reguarity feature only applies to orderless and high-order classes.
Feature	Regularity types of "maximum value" and "minimum value" applies only to number attributes.
Feature	Regularity types of "determined types" and "allowed type" applies only to references.
Feature	A regulated feature assigment must conforme to the regularity feature assignment.
FeatureAssignment	A feature assignment must conforme to the multiplicity and type of its associated feature.

4.5 ML2 Editor

In order to build the models presented here, we employ a language workbench for the specification ML2 models, the ML2 Editor. This editor is built on Xtext[9], a framework for development of textual languages. From an Ecore metamodel and a BNF-like grammar, Xtext delivers an Eclipse-based editor for a textual language. The ML2 Editor, shown in Figure 27, is delivered as an Eclipse plug-in[10], which allows it to be added to any recent release of the environment.

[9] See https://eclipse.org/Xtext/.
[10] Available at https://github.com/claudenirmf/ML2-Editor.



```
module bicycle
{

    // Check
    order 2 class ObjectType isPowertypeOf Object {
        att instancesWeight : [0..1] Number
    };

    // Check
    order 2 class ProductType categorizes Object {
        att instancesPurchasePrice : [0..1] Number
        att instancesSalesPrice : [0..1] Number
    };

    // Check
    class Object {
        att weight : Number
        att color : [0..*] Color
        ref components : [0..*] Component
    };
    // Check
    datatype Color { red:Number green:Number blue:Number };

    /* context BasicComponent inv noComponents: self.components->isEmpty() */
    // Check
    class Component specializes Object;
    // Check
    class BasicComponent specializes Component;
```

3 items

!	Description	Resource	Path	Location	Type
	TODO Discussion: is 'price' a property of individual ob...	Bicycle_challe...	/Bicycle Challenge	line 37	Xtext Task
	TODO Discussion: is certification applicable to types o...	Bicycle_challe...	/Bicycle Challenge	line 123	Xtext Task
	TODO Discussion: this certification applies to certain t...	Bicycle_challe...	/Bicycle Challenge	line 76	Xtext Task

Figure 27 – The ML2 Editor

The ML2 Editor presents many of the features common among modern IDEs, such as *text coloring* (see Figure 28), *auto-completion* (see Figure 29), *refactoring*, *error highlighting* (see Figure 30) and more. In addition, the editor has a validation mechanism that allows for a live verification of ML2's *semantically-motivated syntactic constraints* presented in Table 8, informing the modeler in case of inconsistences on the model.

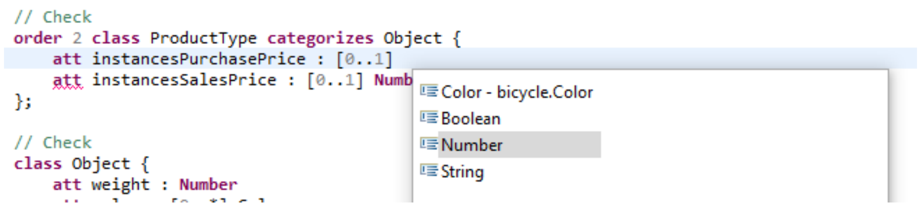

```
// Check
order 2 class ProductType categorizes Object {
    att instancesPurchasePrice : [0..1]
    att instancesSalesPrice : [0..1] Numb
};

// Check
class Object {
    att weight : Number
```

Figure 28 – *Auto-completion* in ML2 models.

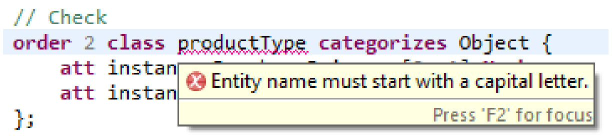

```
// Check
order 2 class productType categorizes Object {
    att instan
    att instan
};
```

Figure 29 – *Error highlighting* in ML2 models.

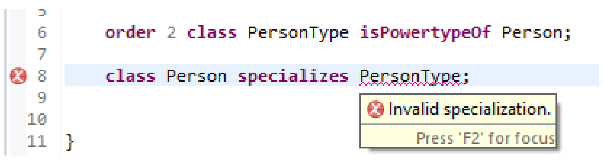

```

    order 2 class PersonType isPowertypeOf Person;

    class Person specializes PersonType;


}
```

Figure 30 – Live validation of *semantically-motivated syntactic constraints*.

Although Xtext is a mature framework for textual language development, its usage impose some implications to the language and its constraints. The first implication affects the metamodel, and the definition of metamodel elements. Since Xtext relies on the instantiation of Ecore objects on each grammar rule, the definition of very complex hierarchies of specialization on the metamodel requires the differentiation of related grammar rules. In other words, additional metaclasses require additional grammar rules to allow their instantiation and, therefore, the usage of additional linguistic elements (e.g., keywords) to differentiate one grammar rule from another. We can see the impact of this characteristic of Xtext on Figure 23, where a specialization of “Feature” into “RegularityFeature” is avoided to reduce language complexity.

The second implication of the usage of Xtext we would like to highlight is related to performance. The built-in options for model validation in Xtext are not always suitable for the more complex constraints present in ML2. Even though we have not performed a test suit for ML2 based on large-scale models, some performance issues were noticed in larger models in the current implementation. This suggests that the built-in validation mechanism of Xtext may not be appropriate for every ML2 constraint and, thus, some other solution for model validation might be needed depending on the user demands.

Chapter 5. Applying ML2

So far, we have presented the Multi-Level Modeling Language highlighting certain aspects of it with small examples. In this chapter, we demonstrate the capabilities of ML2 in more complex scenarios. In Section 5.1, we apply ML2 in the solution of a multi-level challenge proposed for the MULTI 2017 workshop[11]. The challenge consists in the development of a domain conceptual model on product configurations, and was developed independently, preventing bias in the selection of a domain. In Section 5.2, we demonstrate that ML2 is capable of representing the entities in the theory underlying its own semantics. Finally, in Section 5.3, we demonstrate the applicability of ML2 to support the representation of conceptual models in different levels of generality, starting with a foundation ontology called UFO (Unified Foundation Ontology), whose multi-level concepts had been analyzed earlier in terms of MLT (Carvalho *et al.*, 2017).

5.1 The Bicycle Challenge

The Bicycle Challenge is conceptual modeling activity proposed in the context of the 4th International Workshop on Multi-Level Modeling (MULTI 2017 at the MODELS conference). The objective of this challenge is to compare different multi-level modeling approaches on how they solve the same problem. For our purposes, the challenge serves as an external modeling example that can be applied to show ML2 capabilities. Since the Bicycle Challenge consists on capturing the domain description[12], we present portions of the description along with the correspondent model elements. This allows the development of the model through an iterative process, showing how multi-level aspects were identified in the description.

The challenge description starts with the following fragment:

"A configuration is a physical artifact that is composed of components. A component may be composed of other components or of basic parts. There is a difference between the type of a component and its instances. A component has a weight. A bicycle is built of components like frame, a handle bar, two wheels… A bicycle component is a component. A frame, a fork, a wheel, etc. are bicycle components."

Initially, we define the concept "PhysicalObject" in order to capture the physical artifacts prescribed above, as shown in Figure 31. As a physical artifact cannot have instances, and thus "PhysicalObject" is a first-order class. Moreover, "ComplexObject" is a specialization of "PhysicalObject" whose instances are compositions of object, i.e., instances of "Component". The "components" reference relates an instance of "ComplexObject" to the components it may have. The "weight" feature is defined as a feature of "PhysicalObject", rather than

[11] See https://www.wi-inf.uni-duisburg-essen.de/MULTI2017/#challenge.

[12] The Bicycle Challenge is available at https://www.wi-inf.uni-duisburg-essen.de/MULTI2017/.

"Component", as it applies to any physical artifact. The concept "Bicycle" is defined as a specialization "ComplexObject". A number of reference features, subsetting "components", identify the specific components an instance of "Bicycle" may have. Some features, such as "frame" and "fork", use the default multiplicity of ML2 (exactly one, i.e., [1..1]). The types of bicycle components (e.g., "Frame", "Fork", "HandleBar") are specializations of "Component".

```
class PhysicalObject  { att weight : Number };

class ComplexObject specializes PhysicalObject {
       ref components : [1..*] Component
};

class Component specializes PhysicalObject;
class ComplexComponent specializes Component, ComplexObject;

class Bicycle specializes ComplexObject {
       ref frame : Frame subsets components
       ref fork : Fork subsets components
       ref handleBar : HandleBar subsets components
       ref frontWheel : Wheel subsets components
       ref rearWheel : Wheel subsets components
};

class Frame specializes Component;
class Fork specializes ComplexComponent;
class HandleBar specializes Component;
class Wheel specializes Component;
class Suspension specializes Component;
class MudMount specializes Component;
```

Figure 31 – Bicycle Challenge in ML2 (part 1).

Further attributes of elements of domain are provided in the sequel:

"Frames and forks exist in various colors. Every frame has a unique serial number. Front wheel and rear wheel must have the same size. Each bicycle has a purchase price and a sales price."

We promote the feature "color" that is attributed to frames and forks to all physical artifacts and add the *datatype* "Color" to represent particular color values (in the RGB scale). This promotion is not an issue since the assignment of "color" to an instance of "PhysicalObject" is optional. The type "Frame" is added a feature "serialNumber", as shown in the model fragment of Figure 32.

In our interpretation, the domain description deals with product prices in four different ways, leading to the following features: "regularSalesPrice" is the sale price of an individual product in normal conditions; "salesPrice" is the current sale price of an individual product; "purchasePrice" is the actual amount by which an individual product was sold, considering possible discounts and similar factors that may affect its "salesPrice"; finally, "instancesRegularSalesPrice" is the common regular price of all instances a product type (or product model). The first tree features are members of the first-order type "Product", which is a super type of "Bicycle". The feature "instancesRegularSalesPrice" to models of products, i.e., specializations of “Product”, and belongs to the second-order type “ProductType”. The feature “instancesRegularSalesPrice” is, in fact, related to the feature “regularSalesPrice” determining the sale price of all instances of a product type in normal conditions. Therefore, “instancesRegularSalesPrice” regulates the feature “regularSalesPrice” by determining its value.

```
class PhysicalObject {
       att weight : Number
       att color : [0..*] Color
};
datatype Color { red:Number green:Number blue:Number };

order 2 class ProductType categorizes Product {
       regularity instancesRegularSalesPrice : Number
              determinesValue regularSalesPrice
};
class Product {
       att regularSalesPrice : Number
       att salesPrice : Number
       att purchasePrice : Number
};

class Bicycle specializes PhysicalObject, ComplexObject, Product {
       ref frame : Frame subsets components
       ref fork : Fork subsets components
       ref handleBar : HandleBar subsets components
       ref frontWheel : Wheel subsets components
       ref rearWheel : Wheel subsets components
};
class Frame specializes Component, Product {
       att serialNumber : String
};
```

Figure 32 – Bicycle Challenge in ML2 (part 2).

Types of bicycles are introduced in the following fragment:

“There are different types of bicycles for different purposes such as race, mountains, and city. A mountain bike or a city bike may have a suspension. A mountain bike may have a rear suspension. That is not the case for city bikes. A racing fork does not have a suspension. It does not have a mud mount either.”

The description determines specific types of bicycles that are defined according to their intended usage. These specific types of bicycles bring about some others features of bicycle components, as shows Figure 33.

```
class Bicycle specializes PhysicalObject, ComplexObject, Product {
       att suitableForToughTerrains : Boolean
       att suitableForUrbanAreas : Boolean
       att suitableForRacing : Boolean
};

class CityBicycle specializes Bicycle;
class MountainBicycle specializes Bicycle {
       ref rearSuspension : [0..1] Suspension subsets components
};
class RacingBicycle : RacingBicycleType specializes Bicycle;
```

Figure 33 – Bicycle Challenge in ML2 (part 3).

"A racing bike is not suited for tough terrains. A racing bike is suited for races. It can be used in cities, too. Racing frames are specified by top tube, down tube, and seat tube length. A racing bike can be certified by the UCI. A racing frame is made of steel, aluminum, or carbon."

The classifications of components based on their manufacturing materials and usage are orthogonal, allowing the multiple instantiation of these types according to the demand of their instances (e.g., racing frames of aluminum instantiate both "RacingFrame" and "AluminumFrame").

```
class RacingBicycle specializes Bicycle { att isCertified : Boolean };

class RacingFrame specializes Frame {
       att topTubeLength : Number
       att downTubeLength : Number
       att seatTubeLength : Number
};

class SteelFrame specializes Frame;
class AluminumFrame specializes Frame;
class CarbonFrame specializes Frame;

disjoint genset
       general Frame
       specifics SteelFrame, AluminumFrame, CarbonFrame;
```

Figure 34 – Bicycle Challenge in ML2 (part 4).

The challenge's text describes then a last kind of racing bicycle, the "ProRacingBicycle":

"A pro race bike is certified by the UCI. A pro race frame is made of aluminum or carbon. A pro racing bike has a minimum weight of 5200 gr. A carbon frame type allows for carbon or aluminum wheel types only."

This means that the features of a professional racing bicycle, "weight" and "frame", have restricted assignments. Here we make use of a second-order type, "RacingBicycleType", to regulate these features, restricting the minimum weight to 5,2kg and the type of frame component to either aluminum or carbon frames. Notice that bicycle may instantiate multiple professional racing bicycle types, as long as it respect the corresponding regulations on the assignment of it features. Figure 35 presents this fragment of ML2 model.

```
order 2 class RacingBicycleType categorizes RacingBicycle {
       regularity minimumWeight : Number determinesMinValue weight
       regularity ref allowedFrameTypes : [0..*] FrameType
              determinesAllowedTypes frame
};

class ProRacingBicycle :RacingBicycleType specializes RacingBicycle {
       att minimumWeight  = 5.200
       ref allowedFrameTypes =  {AluminumFrame, CarbonFrame}
};

class AluminumWheel specializes Wheel;
class CarbonWheel specializes Wheel;
```

Figure 35 – Bicycle Challenge in ML2 (part 5).

An instance of bicycle types is described:

“*Challenger A2-X is a pro racer for tall cyclists. The regular sales price is 4999.00. Some exemplars are sold for a lower price. It is equipped with a Rocket-A1-XL pro race frame. The Rocket-A1-XL has a weight of 920.0 gr.*”

Finally, Figure 36 highlights the most specific entities from the problem description. Both “ChallengerA2XL” and “RocktA1XL” are defined as first-order classes rather than individuals, because their assignments do not reflect characteristics of individual bicycle components, but characteristics shared by all of their instances. For example, the price of the “ChallengerA2XL” is a feature of all instances of this professional bicycle model. Here we opt to subset the feature “frame”, from “Bicycle”, in order to represent that instances of “ChallengerA2XL” have frame of type “RocketA1XL”. We also add the optional feature “instancesWeight” to “PhysicalObjectType” to facilitate the representation of the weight of instances of component types, as in “RocketA1XL”.

```
class ChallengerA2XL :RacingBicycleType, ProductType specializes ProRacingBicycle {
       att instancesRegularSalesPrice = 4999.00
       ref frame : RocketA1XL subsets frame
};

order 2 class PhysicalObjectType isPowertypeOf PhysicalObject {
       att instancesWeight : [0..1] Number
};

class ProRacingFrame specializes RacingFrame;
class RocketA1XL :ProductType specializes ProRacingFrame {
       att instancesWeight = 0.920
};
```

Figure 36 – Bicycle Challenge in ML2 (part 6).

In order to communicate ML2 models through visual representations, we use from now on a UML profile (OMG, 2011). These visual models are intended to improving the comprehension of the textual representation. However, due to the limitations of UML’s syntax ML2 models can only be partially translated. Table 9 presents a set of stereotypes and the entities they represent in ML2.

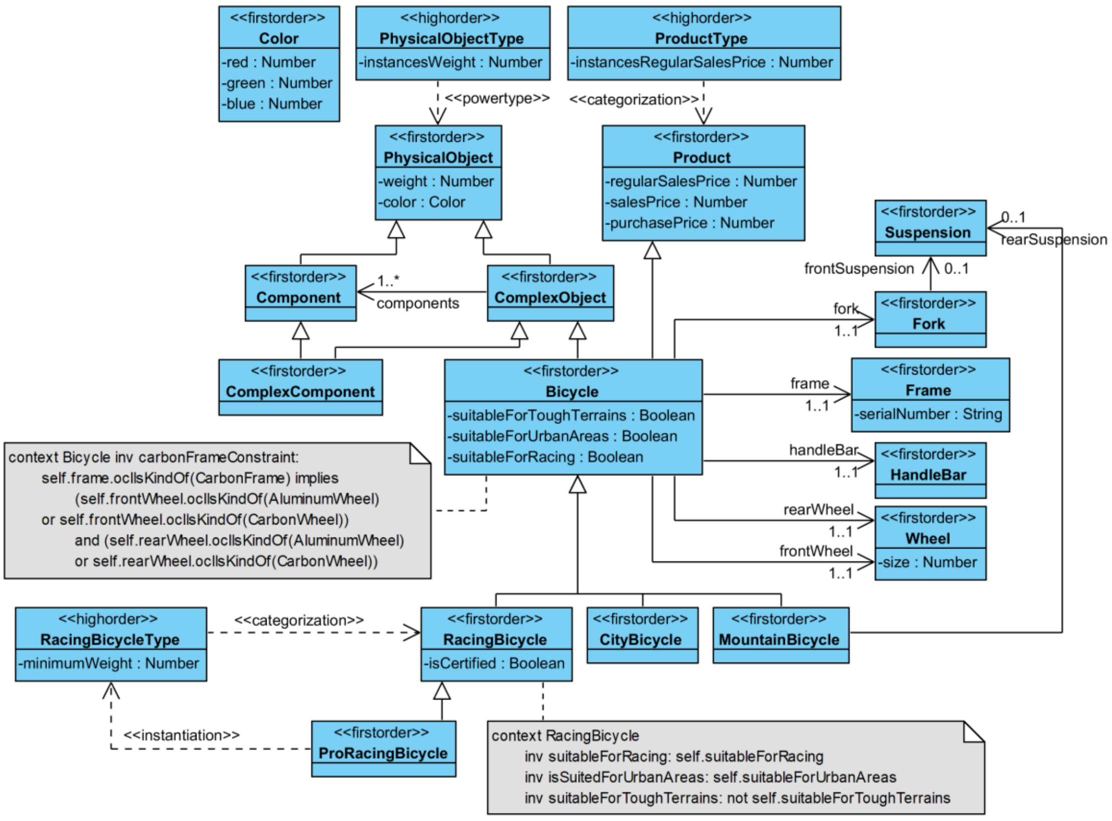
Table 9 – UML profile for ML2.

Stereotype	Metaclass	Semantics
«individual»	Class	A UML Class that represents an individual.
«firstorder»	Class	A UML Class that represents first-order class.
«highorder»	Class	A UML Class that represents a high-order class. It contains a tagged value called 'order' which is an integer corresponding to the class type order.
«orderless»	Class	A UML Class that represents an orderless class.
«instantiation»	Dependency Link	A UML Dependency Link that represents an instantiation relation between the relata. Its equivalent predicate is *instanceof(source,target).*
«subordination»	Dependency Link	A UML Dependency Link that represents a subordination relation between the relata. Its equivalent predicate is *isSubordinatedTo(source,target).*
«powertype»	Dependency Link	A UML Dependency Link that represents a powertype relation between the relata. Its equivalent predicate is *isPowertypeOf(source,target).*
«categorization»	Dependency Link	A UML Dependency Link that represents a categorization relation between the relata. Its equivalent predicate is *categorizes(source,target).*
«completecategorization»	Dependency Link	A UML Dependency Link that represents a complete categorization relation between the relata. Its equivalent predicate is *completelyCategorizes(source,target).*
«disjointcategorization»	Dependency Link	A UML Dependency Link that represents a disjoint categorizationrelation between the relata. Its equivalent predicate is *disjointlyCategorizes(source,target).*
«partitions»	Dependency Link	A UML Dependency Link that represents a partitions relation between the relata. Its equivalent predicate is *partitions(source,target).*

In addition to the stereotypes, some ML2 elements are directly translated the UML constructs. Features are directly translated to UML attributes and directed relationships, while specializations are equivalent between languages. Among the limitations of this profile, there are:

- Individual representation through UML classes: even though UML Classes represent entities that can be instantiated, in order to use individuals in a Class Diagram this artificial transformation is necessary. The importance of it is to bring to the diagram individuals that are relevant to the conceptualization domain (e.g., "Jony" as the designer of the "IPhone5");
- Features assignments are not available: as a two-level scheme modeling language, UML is not able to represent attribute and reference assignments at the class-level;
- Regularity features are not available: no suitable structure was found in UML for representing regularity features.

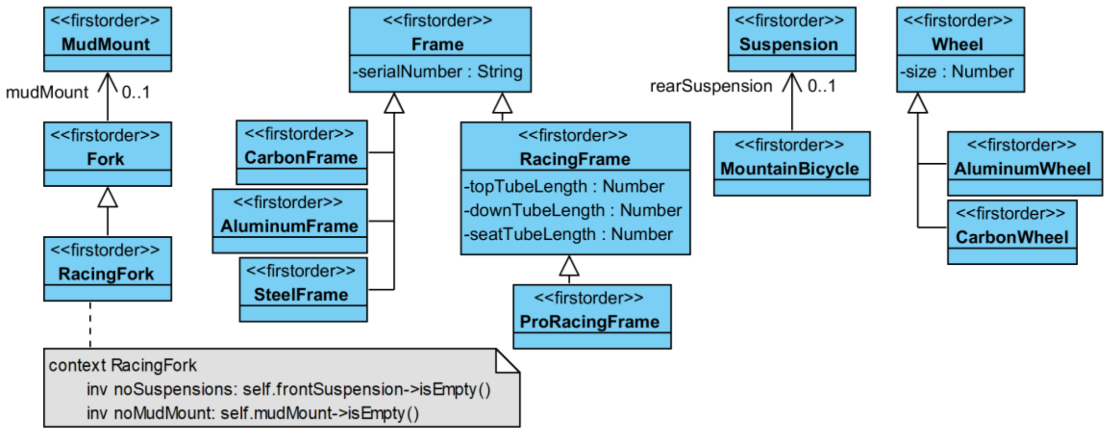
The diagrams in Figure 37 and Figure 38 show a visual representation of the core concepts of the Bicycle Challenge model. In these diagrams, we borrow the OCL (OMG, 2012) syntax with the intention of representing the constraints from the domain rules that could not be captured through ML2 solely. The complete model for the Bicycle Challenge is presented in Appendix C.



```
context Bicycle inv carbonFrameConstraint:
     self.frame.oclIsKindOf(CarbonFrame) implies
          (self.frontWheel.oclIsKindOf(AluminumWheel)
     or self.frontWheel.oclIsKindOf(CarbonWheel))
          and (self.rearWheel.oclIsKindOf(AluminumWheel)
          or self.rearWheel.oclIsKindOf(CarbonWheel))
```

```
context RacingBicycle
     inv suitableForRacing: self.suitableForRacing
     inv isSuitedForUrbanAreas: self.suitableForUrbanAreas
     inv suitableForToughTerrains: not self.suitableForToughTerrains
```

Figure 37 – Bicycle Challenge model (part 1 of 2).



<<firstorder>> MudMount
mudMount
0..1
<<firstorder>> Fork
<<firstorder>> RacingFork
<<firstorder>> Frame
-serialNumber : String
<<firstorder>> CarbonFrame
<<firstorder>> AluminumFrame
<<firstorder>> SteelFrame
<<firstorder>> RacingFrame
-topTubeLength : Number
-downTubeLength : Number
-seatTubeLength : Number
<<firstorder>> ProRacingFrame
<<firstorder>> Suspension
rearSuspension
0..1
<<firstorder>> MountainBicycle
<<firstorder>> Wheel
-size : Number
<<firstorder>> AluminumWheel
<<firstorder>> CarbonWheel

```
context RacingFork
     inv noSuspensions: self.frontSuspension->isEmpty()
     inv noMudMount: self.mudMount->isEmpty()
```

Figure 38 – Bicycle Challenge model (part 2 of 2).

The OCL syntax can also be employed on the elaboration of queries that respond for the demands of a last fragment of the Bicycle Challenge:

> “*A sales manager may be interested in the average sales price of all exemplars of a certain model. He may also be interested in the average sales price of all mountain bikes, all racing bikes etc.*”

This last fragment of the Bicycle Challenge presents competency questions that should be answered by a model designed for the domain description. Figure 39 presents OCL-like queries that illustrate how these competency questions could be addressed in the developed ML2 for the Bicycle Challenge.

```
Query 1: Average sales price of a certain model
let bikePrices : Set(Number) =
        ChallengerA2XL.allInstances()->collect(
                if salesPrice.isOclUndefined()
                then regularSalesPrice else salesPrice endif)
                in
                if bikePrices->notEmpty()
                then bikePrices->sum()/bikePrices->size() else 0.0 endif

Query 2: Average sales price of a certain type of bicycle
let bikePrices : Set(Number) =
        MountainBycicle.allInstances()->collect(
                if salesPrice.isOclUndefined()
                then regularSalesPrice else salesPrice endif)
                in
                if bikePrices->notEmpty()
                then bikePrices->sum()/bikePrices->size() else 0.0 endif
```

Figure 39 – Competency questions of the Bicycle Challenge.

The solution presented here for the Bicycle Challenge shows ML2's capacity for representing general multi-level domains, being able to capture the necessary characteristics of the domain. However, beyond capturing this conceptualization, the challenge also expects the presentation of software solutions based on the conceptual model. In this aspect, ML2 does not provide yet ways for driving the development of software systems other than inspiring developers through a well-defined description of the subject domain. ML2 is also limited in regard to the representation of model constraints, and the design of a proper constraint language for ML2 is an opportunity of future contribution.

5.2 MLT* in ML2

One of the expected applications of ML2 is on the development of foundational models intended for reusability. In this sense, ML2 shows a great capacity on development of models at any degree of generality.

Capturing MLT* in ML2 basically consists on the specification of the constants on the theory (i.e. "Individual", "Entity", "Type", and so on) and the most important relations that can be inferred from their definitions. The resultant models consist on a reflection of the Figure 10, as presented in Figure 40. "Entity" is an instance of "OrderlessClass" that classifies all possible entities. "Class" and "Individual" build the dichotomy between entities that can and cannot have instances, being the former a orderless class that classifies all types of every order, and the later a first-order class that classifies individual entities. Specializing "Class", "OrderedClass" and "OrderlessClass" build another dichotomy between types that fall into a type order and types that defy this ordered structure. Note that both "OrderedClass" and "OrderlessClass" are orderless classes, since their instances spam across different type orders. Instead of capturing a limited number of basic types, in this model we chose to represent the concepts

“FirstOrderClass” and “HighOrderClass”. While “FirstOrderClass” classifies all possible first-order types, “HighOrderClass” classifies every possible ordered type above first-order. Since every entity declaration in ML2 defines the type order of the concept, “HighOrderClass” is sufficient to capture the pattern of basic types from Figure 10. Yet, a further model demands explicit referencing to some specific basic type, i.e., “SecondOrderClass”, “ThirdOrderClass” or beyond, it can be added to the model as an specialization of “HighOrderClass”.

```
module mlt.star
{

      orderless class Entity : OrderlessClass;

      orderless class Class : OrderlessClass specializes Entity
             isPowertypeOf Entity;
      class Individual : FirstOrderClass specializes Entity;

      disjoint complete genset has_instances
             general Entity
             specifics Class, Individual;

      orderless class OrderlessClass : OrderlessClass specializes Class;
      orderless class OrderedClass : OrderlessClass specializes Class;

      disjoint complete genset fixed_order
             general Class
             specifics OrderedClass, OrderlessClass;

      order 2 class FirstOrderClass : HighOrderClass
             specializes OrderedClass
             isPowertypeOf Individual;
      orderless class HighOrderClass : OrderlessClass
             specializes OrderedClass;

      disjoint complete genset high_order
             general OrderedClass
             specifics FirstOrderClass, HighOrderClass;

}
```

Figure 40 – MLT* conceptualizes in ML2 (textual representation).

Figure 41 presents the visual representation of the MLT* model in ML2. The description of MLT* can be reused in models that require explicit reference to the theory’s entities, for example to specify specializations of “Entity” or “Class”.

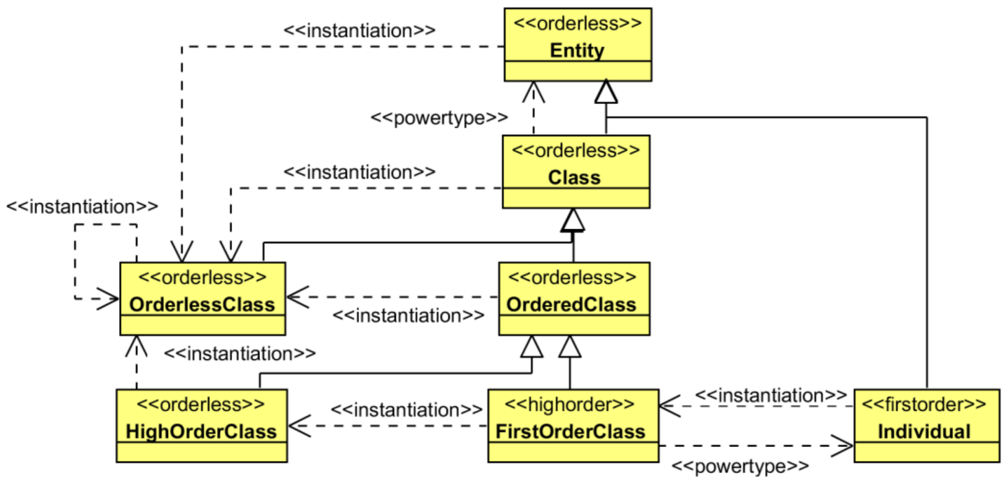


Figure 41 – MLT* conceptualizes in ML2 (diagrammatic representation).

5.3 The Unified Foundation Ontology

The Unified Foundational Ontology (UFO) is a domain independent system of categories aggregating results from disciplines such as Analytical Philosophy, Cognitive Science, Philosophical Logics and Linguistics. Over the years, UFO has been successfully employed to analyze all the classical conceptual modeling constructs including Object Types and Taxonomic Structures, Part-Whole Relations, Intrinsic and Relational Properties, Weak Entities, Attributes and Datatypes, etc. (Guizzardi, 2005). Here we present a portion UFO designed in terms of MLT for multi-level ontology-based conceptual modeling (Carvalho *et al.*, 2017).

UFO works with a distinction of things into *universals* and *individuals*. Universals are predicative entities that apply to a multitude of individuals sharing a set of common features. For example, “John” and “Mary” are both individual instances of the universal “Person”. This classification corresponds to the distinction of things into types and individuals in MLT*. The categories of UFO are compose two taxonomies, the taxonomy of individuals, which defines types of individuals, and the taxonomy of universal, whose instances are types of individuals. This instantiation relation between the taxonomies is evidence of the multi-level nature of UFO.

The classifications over individuals follow with the differentiation between *endurants* and *events*. Instances of “Endurant” are persistent entities that suffer qualitative changes in time keeping its identity (e.g., a house, a person). Endurants are opposed to events, which are sums of temporal parts (e.g. a soccer match, or a meeting). As we focus on structural aspects of UFO, in opposition to dynamic aspects, we follow the UFO taxonomies of endurants and types of endurants, i.e., “Endurant” (see Figure 42) and “EndurantUniversal” (see Figure 45 and Figure 47).

```
class Endurant;

class Substantial specializes Endurant;
class Moment specializes Endurant {
       ref inheresIn : Endurant
};
disjoint complete genset existential_dependece
       general Endurant
       specifics Substantial, Moment;

class Relator specializes Moment;
class IntrinsicMoment specializes Moment;
disjoint complete genset unique_existential_dependence
       general Moment
       specifics IntrinsicMoment, Relator;

class Quality specializes IntrinsicMoment;
class Mode specializes IntrinsicMoment;
disjoint complete genset
       general IntrinsicMoment
       specifics Quality, Mode;

class ExternallyDependentMode specializes Mode {
       ref partOf : Relator
       ref externallyDepedentOn : [1..*] Endurant
};
```

Figure 42 –Taxonomy of “Endurant” in ML2.

Figure 43 and Figure 44 present a visual representation of the taxonomy of “Endurant” of UFO.

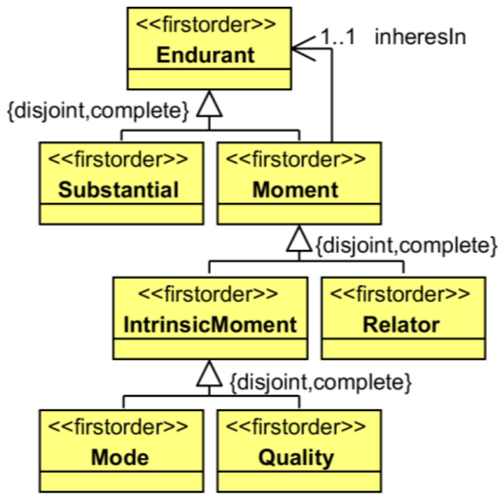


Figure 43 – Visual representation of the taxonomy of “Endurant”.

Endurants are further classified into “Substantial” and “Moments”. Instances of “Substantial” are existentially independent endurants (e.g. a person, an organization). Instances of “Moment”, on the other hand, are property-like entities that inhere in another endurant, therefore, being existentially dependent on that endurant (e.g., an object’s color, an enrollment). Individual moments are further classified according to the sort of endurant they inhere in. Moments that inhere on one single endurant are instances of “IntrinsicMoment” (e.g. a person's age or height, a thought, a belief). In opposition to intrinsic moments, relational moments (moments that inhere in a sum of endurants) are instances of "Relator" (e.g. a marriage, an employment, an enrollment).

Intrinsic moments in UFO are further classified into "Quality" and "Mode". The instances of "Quality" are measurable and can be directly related to some quality structure (e.g., the color of an object that can be described on the RGB or HSV scales). In contrast, non-measurable intrinsic moments are instances of "Mode" (e.g., a person's skills, intentions, beliefs or symptoms). In addition, UFO also accounts for a special kind of mode whose instances capture individual properties of endurants involved in a relational moment, the "ExternallyDependentMode", shown in Figure 44. Since an externally dependent moment exists solely on the context of a relation, it is part of a relator, having an external existential dependency on all others endurants involved in the relation. An example of this kind of mode is the commitment of a husband or a wife involved in a marriage.

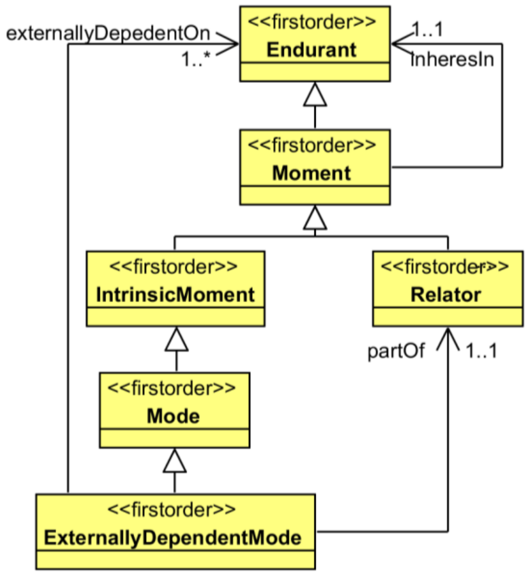


Figure 44 – Classifications of moments in UFO.

Besides the taxonomy of endurants, UFO also defines a taxonomy for the classification of types of endurants, also called endurant universals. The topmost concept of this taxonomy is "EndurantUniversal", a second-order type whose instances classify endurant individuals (e.g., while "Person" is an instance of "EndurantUniversal", the person "John" is an instance of "Endurant"). As shows Figure 46 (visual representation of the model fragment in Figure 45), every endurant type on the taxonomy presented in Figure 43 has an equivalent second-order class. These universal types have as instances subtypes of their non-universal equivalents, therefore, categorizing their equivalents.

```
order 2 class EndurantUniversal categorizes Endurant;

order 2 class SubstantialUniversal specializes EndurantUniversal
       categorizes Substantial;
order 2 class MomentUniversal specializes EndurantUniversal
       categorizes Moment;

disjoint complete genset existential_dependece_of_instances
       general EndurantUniversal
       specifics SubstantialUniversal, MomentUniversal;

order 2 class RelatorUniversal specializes MomentUniversal
       categorizes Relator;
order 2 class IntrinsicMomentUniversal specializes MomentUniversal
       categorizes IntrinsicMoment;

disjoint complete genset unique_existential_dependence_of_instances
       general MomentUniversal
       specifics IntrinsicMomentUniversal, RelatorUniversal;

order 2 class QualityUniversal specializes IntrinsicMomentUniversal
       categorizes Quality;
order 2 class ModeUniversal specializes IntrinsicMomentUniversal
       categorizes Mode;

disjoint complete genset
       general IntrinsicMomentUniversal
       specifics QualityUniversal, ModeUniversal;
```

Figure 45 –Taxonomy of Endurant Universal in ML2.

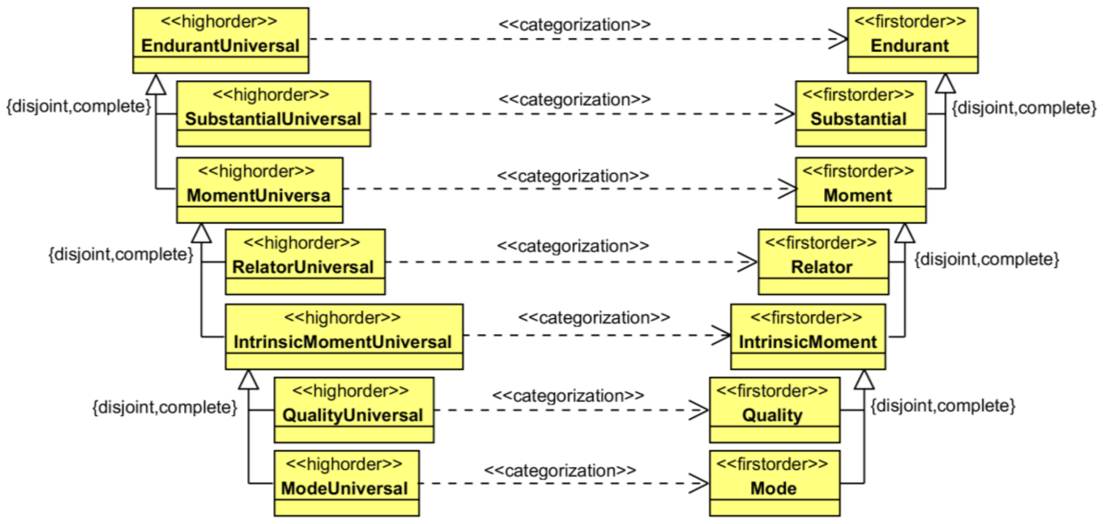


Figure 46 – Specializations of “EndurantUniversal” categorizing specialization of “Endurant”.

Additionally, UFO defines further specializations of “SubstantialUniversal” according to ontological notions of identity and rigidity (see Figure 47). Substantial universals that carry a uniform principle of identity for their individuals are instances of “SortalUniversal” (e.g., “Person”, “Car”, “Organization”). In contrast, instances of “MixinUniversal” represent an abstraction of properties that are common to instances of various sortals (e.g., the mixin “Insurable Item” describes properties that are common to entities of different sortals such as “House”, “Car”, “Work of Art”).

```
order 2 class MixinUniversal specializes SubstantialUniversal;
order 2 class SortalUniversal specializes SubstantialUniversal;
disjoint complete genset
       general SubstantialUniversal
       specifics SortalUniversal, MixinUniversal;

order 2 class RigidMixin specializes MixinUniversal;
order 2 class AntiRigidMixin specializes MixinUniversal;
disjoint complete genset
       general MixinUniversal
       specifics RigidMixin, AntiRigidMixin;

order 2 class Category specializes RigidMixin;
order 2 class PhaseMixin specializes AntiRigidMixin;
order 2 class RoleMixin specializes AntiRigidMixin;
disjoint complete genset
       general AntiRigidMixin
       specifics PhaseMixin, RoleMixin;

order 2 class RigidSortal specializes SortalUniversal;
order 2 class AntiRigidSortal specializes SortalUniversal;
disjoint complete genset
       general SortalUniversal
       specifics RigidSortal, AntiRigidSortal;

order 2 class Kind specializes RigidSortal partitions Substantial;
order 2 class Subkind specializes RigidSortal subordinatedTo Kind;
disjoint complete genset
       general RigidSortal
       specifics Kind, Subkind;

order 2 class Phase specializes AntiRigidSortal subordinatedTo Kind;
order 2 class Role specializes AntiRigidSortal subordinatedTo Kind;
disjoint complete genset
       general AntiRigidSortal
       specifics Phase, Role;
```

Figure 47 –Further specializations of "Substantial Universal".

Moreover, UFO defines subtypes of "SortalUniversal" and "MixinUniversal" based on the rigidity of the endurant types they classify, lead to a rigid and an anti-rigid subtype of each. A rigid type classifies its instances necessarily, i.e., an instance of a rigid type remains its instance as long as it exists. On the other hand, an anti-rigid type classifies entities that are not necessarily instantiates it whenever it is present. For example, an instance of "Person" remains a person during its entire exists, while a person turn into and cease to be an instance of "Student" through time.

Both "RigidSortal" and "RigidMixin" also have subtypes based on the criteria used by its instances to classify endurants. Instances of "Phase" and "PhaseMixin" are types defined based on intrinsic properties of their instances, e.g., "Person" may have subtypes "Child" and "Adult", instances of "Phase", that classify instances of "Person" based on the intrinsic property age. Opposite to phases, instances of "Role" and "RoleMixin", are types that classify endurants based on relational properties, as in the case of "Husband" and "Wife", subtypes of "Person" that classify entities involved in a marriage relator.

Rigid sortals, rigid types whose instances carry a uniform principle of identity, are differentiated into “Kind” and “Subkind”. The instances of “Kind” are the most general sortals of their hierarchies since they provide the identity principle to its instances (e.g., “Person”, “Organization”, “House”). Every sortal must specialize one instance of “Kind” that represents the uniform identity of the entities it classifies. The instances of “Subkind”, equally to instances of “Kind”, necessarily classify entities that share some uniform identity, however they do not provide this identity (e.g., “Man”, “Women”, “NonProfitOrganization”). Rigid mixins that represent types that are necessarily applied to types of different kinds are instances of “Category” (e.g., “Legal Entity”, whose instances can be persons or organizations, entities of different sorts).

Finally, two rules defined in UFO are represented in the model of Figure 47 through MLT* relations. The partitions relation from “Kind” to “Substantial” represents that every instance of the latter instantiates one instance of the former, i.e., every instance of “Substantial” most have an identity supplied by an instance of “Kind”. Furthermore, every sortal must specialize one instance of “Kind” in order to classify entities that carry a uniform identity, thus, “Subkind”, “Phase” and “Role” are subordinated to “Kind”. These categories of types of substantial are also presented in Figure 48 in a visual representation.

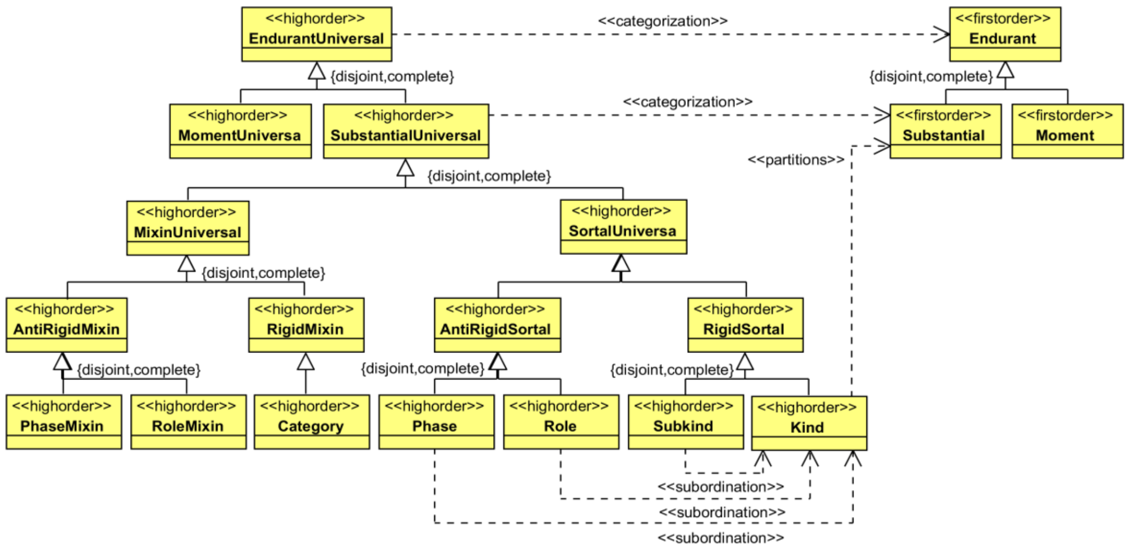


Figure 48 – Visual representation of specializations of “Substantial Universal”.

The work of Carvalho *et al.* (2017) presents principles for the development of multi-level ontologies founded in UFO. The strategy presented by the authors can be used on the design of hierarchies of models that specialize and instantiate the concepts of UFO. Through the model presented in this section, ML2 can be used as representation language on the development of such model, even allowing domain models founded in both UFO and MLT*, by using the model from previous section.

This section only uses diagrams to represent UFO models, however, the full textual model in ML2 can be seen on the Appendix D.

Chapter 6. Final Considerations

In this dissertation, we have presented the ML2 multi-level conceptual modeling language. We have approached the design of ML2 with a careful consideration of the conceptualization of types in classification schemes that transcend a rigid two-level structure. The conceptualization was formalized in a theory called MLT* (Chapter 3). We have aimed for a simple but comprehensive theory that encompasses stratified and non-stratified schemes, and is able to accommodate the variations for the powertype pattern in the literature.

The theory's elements and rules were reflected in the constructs of ML2 (Chapter 4). The language was designed to offer expressiveness for the domain modeler by addressing a set of comprehensive representation requirements (Chapter 2). Further, rules incorporated in the language have been implemented in a featured Eclipse-based editor that supports the live verification of models to ensure adherence to the underlying theory. Lastly, we have shown the language's capabilities on capturing multi-level domains from different sources, which are either proposed by the community[13] or relevant in the literature (Carvalho *et al.*, 2017).

We have observed in the literature that multi-level approaches often opt for one of two extremes: (i) to consider all classes to be orderless (or similarly to ignore the organization of elements into stratified orders/levels altogether, what is referred to as a level-blind approach in (Atkinson, Gerbig and Kühne, 2014)), or (ii) to consider all classes to be strictly stratified. Approaches that opt for (i) are able to represent all types which can be captured by ML2, however, fail to provide rules to guide the use of the various structural relations (including instantiation). As shown in (Brasileiro *et al.*, 2016a), this lack of guidance has serious consequences for the quality of the resulting representation. Approaches that opt for the other end of the spectrum (ii) do not support the representation of a number of important abstract notions, including those very general notions that we use to articulate multi-level domains (such as "types", "clabjects", "entities"). The combination of both approaches in the design of ML2 places it in a unique position in multi-level modeling approaches.

A few other knowledge representation approaches (such as Telos (Jarke *et al.*, 1995) and Cyc (Foxvog, 2005)) have, like ML2, drawn distinctions between orderless and ordered types. Differently from ML2, however, Telos does not provide rules for the various structural relations, including instantiation and specialization. In its turn, Cyc, which is currently the world's largest and arguably most mature knowledge base, employs a conceptual architecture for types that is most similar to MLT*'s top layer. Similarly to MLT*, this architecture includes

[13] Available at https://www.wi-inf.uni-duisburg-essen.de/MULTI2017/#challenge.

rules for instantiation and specialization (Foxvog, 2005). Differently from ML2, however, the rules are not incorporated in the representation language and no deep characterization mechanism is provided.

6.1 Contributions

Here we present a list of the specific contributions of this work:

- **A reviewed set of requirement for multi-level conceptual modeling:** by revisiting an earlier proposal for a set of requirements for multi-level modeling (Brasileiro *et al.*, 2016b) and analyzing a collection of prominent approaches in the literature, we define a revised set of requirements for multi-level conceptual modeling (satisfying specific objective O1). This revision aims to accommodate domain entities that do not fall into a specific type order and to account for the treatment of entity features in multi-level domains. This set of requirements drives the definition of our theory and language, and also serves as basis for comparison with existing approaches to multi-level modeling;
- **A revision of the MLT Multi-Level Theory**: also by revisiting an earlier proposal, in Chapter 3 we have generalized the theory proposed by Carvalho and Almeida (2016). This allowed the definition of MLT* (satisfying O2), a multi-level theory able to handle entities that do not fall into a specific type order. As MLT, MLT* can be considered a reference ontology for types in multi-level conceptual modeling, being used to inform the development of ML2 as a well-founded language for multi-level conceptual modeling. This contribution was the result of joint work conducted with the supervisors João Paulo A. Almeida and Victorio A. de Carvalho and reported in (Almeida, Fonseca and Carvalho, 2017);
- **Design of a Multi-Level Modeling Language**: based on MLT*, we define a textual language for multi-level conceptual modeling dubbed ML2. This language is suited for conceptual modeling in the sense it aims at fulfilling the requirements defined in Section 2.4 (satisfying O3). The language is supported by an Eclipse-based language workbench, which provides productivity tools for the development of ML2 models, including live validation mechanisms (satisfying O4). The validation of ML2 models ensures their compliance to the rules defined in MLT*. In addition, a selected number of models are presented in Chapter 5, demonstrating the language's capacity for handling models at different degrees of generality (satisfying O5).

In the scope of this master dissertation, three papers have been accepted or published in peer-reviewed conferences and journals:

ALMEIDA, João Paulo A.; FONSECA, Claudenir Morais; CARVALHO, Victorio Albani. **A Comprehensive Formal Theory for Multi-Level Conceptual Modeling.** In: 36th International Conference on Conceptual Modeling, ER 2017 (*accepted*).

CARVALHO, Victorio Albani; ALMEIDA, João Paulo A.; FONSECA, Claudenir Morais; GUIZZARDI, Giancarlo. **Extending the Foundations of Ontology-based Conceptual Modeling with a Multi-Level Theory.** In: Proceedings of the 35th International Conference on Conceptual Modeling, ER 2015. DOI: 10.1007/978-3-319-25264-3_9. (Best Paper Award)

CARVALHO, Victorio Albani; ALMEIDA, João Paulo A.; FONSECA, Claudenir Morais; GUIZZARDI, Giancarlo. **Multi-Level Ontology-based Conceptual Modeling.** Data & Knowledge Engineering (accepted).

The first paper (Almeida, Fonseca and Carvalho, 2017) presents the proposed extension of MLT. It contains both the core concepts of MLT* and the set of requirements that drove its development. The second and third papers, (Carvalho *et al.*, 2015) and (Carvalho *et al.*, 2017), are also in the context of the MLT theory, being the third an extension of the second. In (Carvalho *et al.*, 2015), MLT is employed to approach the development of UFO-based ontologies in a multi-level domain. (Carvalho *et al.*, 2017) expands the previous work by addressing also the definition of types of properties (*moments*), an issue originally out of the scope of (Carvalho *et al.*, 2015), which dealt only with rules for types of *substantials*. Both (Carvalho *et al.*, 2015) and (Carvalho *et al.*, 2017) contain a version of UFO based on MLT, which was used in Section 5.3 as a selected case for presenting the ML2 language.

6.2 Future Work

Some research opportunities arise from the proposals of this work:

- **Development of transformations from ML2**: prior to ML2, Brasileiro *et al.* (2016b) proposed a schema for the representation of MLT entities in the Semantic Web. Even though MLT*-based models may account for entities that cannot fit into MLT, ML2 models could be fully transformed into a revised version of this Semantic Web scheme. This step would benefit the development of Semantic Web applications from ML2 models;
- **Development of an integrated constraint language**: on the design of the Bicycle Challenge model, we have employ the OCL syntax (OMG, 2012) to capture constraints of the given domain that cannot be directly represented in the modeling language. This necessity showed a limitation on the expressivity of ML2 which may be addressed through an integrated constraint language. The

suitability of OCL and other constraint languages for this task is a subject of further investigation;

- **Investigation of a suitable visual syntax to accompany the textual syntax**: although we provide a profile for the representation of ML2 models in UML, this representation is partial, as UML does not have mechanisms that allow the translation of all elements of an ML2 model. For example, UML does not allow assignment of values of features of instances of a high-order type. Moreover, as it was not part of our scope, we have not considered other techniques for visual representation of MLT models, such as (Carvalho, Almeida and Guizzardi, 2016). The importance of visual representation of these models is the capacity of visual languages as means of communication;
- **Interpretation of attributes and relations in light of MLT***: in the context of this dissertation we were not able to revisit original MLT's support for attributes, relations and their regularities in MLT*. Thus, we rely on the original work (Carvalho and Almeida, 2016) to convey semantics to our treatment of features. This is particularly important to account for the deep characterization mechanisms in potency-based approaches (Atkinson and Kühne, 2008). Therefore, a future work should advance the theory on this topic.
- **Enhancements in other approaches based on the investigation of the implications of MLT***: as discussed in (Guizzardi, 2005), a reference theory can be used to inform the revision and redesign of a modeling language, not only through the identification of semantic overload, construct deficit, construct excess and construct redundancy, but also through the definition of modeling patterns and semantically-motivated syntactic constraints. Thus, natural applications for MLT* include: (i) its use in the design of well-founded multi-level conceptual modeling languages (as shown here for ML2); (ii) it is used in the redesign of a (non-multi-level) modeling language such as UML (Carvalho, Almeida and Guizzardi, 2016) or (iii) the analysis and redesign of existing multi-level modeling languages. Concerning the latter, in Section 2.5, we have presented an analysis of existing multi-level modeling languages, identifying the requirements not addressed by those languages. We believe that MLT* can be used to establish proposals for revision and extension of these approaches to improve their expressivity and enable the representation of domains they cannot currently address.

References

Almeida, J. P. A., Fonseca, C. M. and Carvalho, V. A. (2017) “A Comprehensive Formal Theory for Multi-Level Conceptual Modeling,” in *ER Conferece 2017*. Valencia, Spain.

Atkinson, C. and Gerbig, R. (2012) “Melanie: multi-level modeling and ontology engineering environment,” *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards*, pp. 5–6.

Atkinson, C., Gerbig, R. and Kühne, T. (2014) “Comparing multi-level modeling approaches,” in *Proceedings MULTI 2014: 1st International Workshop on Multi-Level Modelling, co-located with MODELS 2014*, pp. 53–61.

Atkinson, C. and Kuhne, T. (2001) “Processes and Products in a Multi-level Metamodeling Architecture,” *Int. Journal of Software Engineering and Knowledge Engineering*, 11(6), pp. 761–784.

Atkinson, C. and Kühne, T. (2000) “Meta-level Independent Modeling,” in *International Workshop “Model Engineering” (in conjunction with ECOOP’2000)*. Cannes, France, p. 16.

Atkinson, C. and Kühne, T. (2003) “Model-driven development: A metamodeling foundation,” *IEEE Software*, 20(5), pp. 36–41.

Atkinson, C. and Kühne, T. (2008) “Reducing accidental complexity in domain models,” *Software & Systems Modeling*, 7(3), pp. 345–359.

Bagaria, J. (2017) “Set Theory,” in Zalta, E. N. (ed.) *The Stanford Encyclopedia of Philosophy*. Summer 201. Metaphysics Research Lab, Stanford University.

Brasileiro, F. *et al.* (2016a) “Applying a Multi-Level Modeling Theory to Assess Taxonomic Hierarchies in Wikidata,” in *Proceedings of the 25th International Conference Companion on World Wide Web*. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee (WWW ’16 Companion), pp. 975–980.

Brasileiro, F. *et al.* (2016b) “Expressive Multi-level Modeling for the Semantic Web,” in Groth, P. et al. (eds.) *The Semantic Web -- ISWC 2016: 15th International Semantic Web Conference, Kobe, Japan, October 17--21, 2016, Proceedings, Part I*. Cham: Springer International Publishing, pp. 53–69.

Cardelli, L. (1988) “Structural subtyping and the notion of power type,” *Proceedings of the 15th ACM SIGPLANSIGACT symposium on Principles of programming languages POPL 88*. New York, New York, USA: ACM Press, pp. 70–79.

Carvalho, V. A. *et al.* (2015) “Extending the Foundations of Ontology-Based Conceptual Modeling with a Multi-level Theory,” in Johannesson, P. et al. (eds.) *Conceptual Modeling: 34th International Conference, ER 2015, Stockholm, Sweden, October 19-22, 2015, Proceedings*. Cham: Springer International Publishing, pp. 119–

133.

Carvalho, V. A. *et al.* (2017) “Multi-level ontology-based conceptual modeling,” *Data and Knowledge Engineering*. Elsevier B.V., 109(March), pp. 3–24.

Carvalho, V. A. and Almeida, J. P. A. (2015) “A Semantic Foundation for Organizational Structures: A Multi-level Approach,” *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOCW*, 2015–Novem, pp. 50–59.

Carvalho, V. A. and Almeida, J. P. A. (2016) “Towards a Well-Founded Theory for Multi-Level Conceptual Modeling,” *Software {&} Systems Modeling*.

Carvalho, V. A., Almeida, J. P. A. and Guizzardi, G. (2016) “Using a well-founded multi-level theory to support the analysis and representation of the powertype pattern in conceptual modeling,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pp. 309–324.

Chen, P. P.-S. (1976) “The entity-relationship model: Towards a unified view of data,” *ACM Trans. Database Syst.* New York, NY, USA: ACM, 1(1), pp. 9–36.

Clark, T., Gonzalez-Perez, C. and Henderson-Sellers, B. (2014) “Foundation for multi-level modelling,” *CEUR Workshop Proceedings*, 1286, pp. 43–52.

ECMA (2013) *The JSON Data Interchange Format*. 1st Editio. Available at: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

Fowler, M. (1996) *Analysis Patterns: Reusable Object Models*. 1st ed. Addison-Wesley Professional.

Foxvog, D. (2005) “Instances of instances modeled via higher-order classes,” *Foundational Aspects of Ontologies*, (9–2005), pp. 46–54. Available at: http://www.uni-koblenz.de/fb4/publikationen/gelbereihe/RR-9-2005.pdf.

Frank, U. (2014) “Multilevel Modeling,” *Business & Information Systems Engineering*, 6(6), pp. 319–337.

Gonzalez-Perez, C. and Henderson-Sellers, B. (2006) “A powertype-based metamodelling framework,” *Software and Systems Modeling*, 5(1), pp. 72–90.

Guarino, N. and Guizzardi, G. (2015) *“We need to Discuss the Relationship”: Revisiting Relationships as Modeling Constructs, 27th International Conference on Advanced Information Systems Engineering (CAISE 2015)*.

Guizzardi, G. (2005) *Ontological Foundations for Structural Conceptual Models*. 1st ed. The Netherlands.

Guizzardi, G. *et al.* (2015) “Towards an Ontological Analysis of Powertypes,” *Proceedings of the Joint Ontology Workshops 2015*, 1517.

Henderson-Sellers, B. (2012) *On the Mathematics of Modelling, Metamodelling, Ontologies and Modelling Languages On the Mathematics of Modeling, Metamodelling, Ontologies and Modelling Languages*.

Igamberdiev, M. *et al.* (2016) “An integrated multi-level modeling approach for industrial-scale data interoperability,” *Software and Systems Modeling*, pp. 1–26.

Igamberdiev, M., Grossmann, G. and Stumptner, M. (2014) “An implementation of multi-level modelling in F-Logic,” *CEUR Workshop Proceedings*, 1286(1), pp. 33–42.

Irvine, A. D. and Deutsch, H. (2016) “Russell’s Paradox,” in Zalta, E. N. (ed.) *The Stanford Encyclopedia of Philosophy*. Winter 201. Metaphysics Research Lab, Stanford University.

Jackson, D. (2006) *Software Abstractions: Logic, Language and Analysis*. MIT PRess.

Jarke, M. *et al.* (1995) “ConceptBase - A deductive object base for meta data management,” *Journal of Intelligent Information Systems*, 4(2), pp. 167–192.

Jeusfeld, M. A. and Neumayr, B. (2016) “DeepTelos: Multi-level Modeling with Most General Instances,” in *35th International Conference, ER 2016*. Springer International Publishing, pp. 198–211.

Kuehne, T. and Schreiber, D. (2007) “Can programming be liberated from the two-level style?,” *ACM SIGPLAN Notices*, 42(10), pp. 229–244.

Lara, J. *et al.* (2013) “Extending Deep Meta-Modelling for Practical Model-Driven Engineering,” *The Computer Journal*.

de Lara, J. and Guerra, E. (2010) “Deep Meta-modelling with MetaDepth,” in *Proc. of the 48th International Conference, TOOLS 2010*. Málaga, Spain, pp. 1–20.

Lara, J. De, Guerra, E. and Cuadrado, J. S. J. S. (2014) “When and How to Use Multilevel Modelling,” *ACM Trans. Softw. Eng. Methodol.*, 24(2), pp. 1–46.

Masolo, C. *et al.* (2003) “Ontology Library,” *ICT project*, 33052, p. 343. Available at: http://wonderweb.semanticweb.org/deliverables/documents/D18.pdf.

Mayr, E. (1982) *The Growth of Biological Thought: Diversity, Evolution, and Inheritance*. The Belknap Press.

Menzel, C. (2011) “Knowledge representation, the World Wide Web, and the evolution of logic,” *Synthese*, 182(2), pp. 269–295.

Mylopoulos, J. *et al.* (1990) “Telos: Representing Knowledge About Information Systems,” *ACM Trans. Inf. Syst.* New York, NY, USA: ACM, 8(4), pp. 325–362.

Mylopoulos, J. (1992) “Conceptual modeling and Telos,” *Conceptual Modelling, Databases, and CASE: an Integrated View of Information System Development, John Wiley &amp; Sons, New York, New York*. Edited by P. Loucopoulos and R. Zicari. Wiley, pp. 49–68.

Neumayr, B. *et al.* (2014) “Dual Deep Instantiation and Its ConceptBase Implementation,” in *International Conference on Advanced Information Systems Engineering*, pp. 503–517.

Neumayr, B., Grün, K. and Schrefl, M. (2009) “Multi-level domain modeling with m-objects and m-relationships,” *Conferences in Research and Practice in Information Technology Series*. Wellington, New Zealand, 96(Apccm).

Odell, J. (1994) “Power Types,” *Journal of Object-Oriented Programing*, 7(2), pp. 8–12.

OMG (2011) “UML Superstructure v2.4.1.” Available at: http://www.omg.org/spec/UML/2.4.1/.

OMG (2012) “OMG Object Constraint Language (OCL) - Version 2.3.1.”

Selway, M. et al. (2017) “A conceptual framework for large-scale ecosystem interoperability and industrial product lifecycles,” *Data and Knowledge Engineering*, 109(March), pp. 85–111.

W3C (2009) *OWL 2 Web Ontology Language Document Overview*. Available at: http://www.w3.org/TR/2009/REC-owl2-overview-20091027/.

W3C (2014) “RDF 1.1 concepts and abstract syntax.” World Wide Web Consortium.

Yang, G. *et al.* (2005) “FLORA-2: User’s manual,” *Version 0.94 (Narumigata). April*.

Appendix A. Specification of ML2's Abstract Syntax in Xcore

This appendix presents a specification of the metamodel (abstract syntax) of ML2. This metamodel is specified in Xcore, a textual syntax for Ecore models.

```
@Ecore(nsURI="http://www.nemo.inf.ufes.br/ml2/ML2")
@GenModel(modelDirectory="br.ufes.inf.nemo.ml2.meta/src-gen")
package br.ufes.inf.nemo.ml2.meta

import org.eclipse.emf.common.util.BasicEList

class ML2Model
{
	String name
	refers ML2Model[] includes
	contains ModelElement[] elements
}

abstract class ModelElement {}

class Import extends ModelElement
{
	String importedNamespace
}

abstract class EntityDeclaration extends ModelElement
{
	String name
	refers ML2Class[] instantiatedClasses

	contains FeatureAssignment[] assignments

	op boolean isUnnamed()
	{
		return name===null || name==""
	}
}

class Individual extends EntityDeclaration {}

abstract class ML2Class extends EntityDeclaration
{
	refers ML2Class[] superClasses
	refers ML2Class[] subordinators
	refers ML2Class powertypeOf
	refers ML2Class categorizedClass
	CategorizationType[0..1] categorizationType

	contains Feature[] features

	op Attribute[] getAttributes()
	{
		val l = new BasicEList<Attribute>()
		features.forEach[if(it instanceof Attribute) l.add(it)]
		return l
	}
```

```
       op Reference[] getReferences()
       {
              val l = new BasicEList<Reference>()
              features.forEach[if(it instanceof Reference) l.add(it)]
              return l
       }
}

enum CategorizationType
{
       CATEGORIZER as "categorizes"
       DISJOINT_CATEGORIZER as "disjointCategorizes"
       COMPLETE_CATEGORIZER as "completeCategorizes"
       PARTITIONER as "partitions"
}

class OrderlessClass extends ML2Class {}

abstract class OrderedClass extends ML2Class {}

class HOClass extends OrderedClass
{
       Integer order
}

class FOClass extends OrderedClass {}

class DataType extends FOClass {}

class GeneralizationSet extends ModelElement
{
       String name = "anonymous"
       boolean isDisjoint = "false"
       boolean isComplete = "false"
       refers ML2Class general
       refers ML2Class categorizer
       refers ML2Class[] specifics
}

class Feature extends ModelElement
{
       String name
       int lowerBound = "1"
       int upperBound = "1"

       RegularityFeatureType regularityType
       refers Feature regulatedFeature
}

enum RegularityFeatureType
{
       DETERMINES_MAX_VALUE as "determinesMaxValue"
       DETERMINES_MIN_VALUE as "determinesMinValue"
       DETERMINES_VALUE as "determinesValue"
       DETERMINES_ALLOWED_VALUES as "determinesAllowedValues"
       DETERMINES_TYPE as "determinesType"
       DETERMINES_ALLOWED_TYPES as "determinesAllowedTypes"
}

class Attribute extends Feature
{
       PrimitiveType primitiveType
       refers DataType _type
       refers Attribute[] subsetOf
```

```
	op boolean isPrimitive()
	{
		return !eIsSet(MetaPackage.eINSTANCE.attribute__type)
	}
}

class Reference extends Feature
{
	refers ML2Class _type
	refers Reference[] subsetOf
	refers Reference oppositeTo
}

enum PrimitiveType
{
	STRING as "String"
	NUMBER as "Number"
	BOOLEAN as "Boolean"
}

class FeatureAssignment extends ModelElement {}

class AttributeAssignment extends FeatureAssignment
{
	refers Attribute attribute
	refers Individual[] individualAssignments
	contains Individual[] unnamedIndividualAssignments
	contains Literal[] literalAssignments

	op boolean hasIndividualAssignments()
	{
		return !individualAssignments.isEmpty ||
!unnamedIndividualAssignments.isEmpty
	}
	op boolean hasLiteralAssignments()
	{
		return !literalAssignments.isEmpty
	}
	op Individual[] getAllIndividualAssignments()
	{
		val l = new BasicEList<Individual>
		l.addAll(individualAssignments)
		l.addAll(unnamedIndividualAssignments)
		return l
	}
	op Object[] getAllAssignments()
	{
		val l = new BasicEList<Object>
		l.addAll(individualAssignments)
		l.addAll(unnamedIndividualAssignments)
		l.addAll(literalAssignments)
		return l
	}
}

class ReferenceAssignment extends FeatureAssignment
{
	refers Reference reference
	refers EntityDeclaration[] assignments
}

abstract class Literal extends ModelElement {}
```

```
class ML2String extends Literal
{
        String value

        op boolean equals(Object obj)
        {
                if(obj instanceof ML2String) return value == obj.value
                else    return super.equals(obj)
        }
}

class ML2Number extends Literal
{
        double value

        op boolean equals(Object obj)
        {
                if(obj instanceof ML2Number) return value === obj.value
                else    return super.equals(obj)
        }
}

class ML2Boolean extends Literal
{
        boolean value

        op boolean equals(Object obj)
        {
                if(obj instanceof ML2Boolean)       return value === obj.value
                else    return super.equals(obj)
        }
}
```

Appendix B. Specification of ML2's Concrete Syntax in Xtext Grammar

This appendix presents a specification of ML2's concrete syntax in Xtext, a BNF-like grammar that directly references the elements of the associated metamodel.

```
grammar br.ufes.inf.nemo.ml2.ML2 with org.eclipse.xtext.common.Terminals

import "http://www.nemo.inf.ufes.br/ml2/ML2"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

ML2Model:
       'module' name=QualifiedName '{'
       ('include' includes+=[ML2Model|QualifiedName] ';' | elements+=ModelElement)*
       '}';

QualifiedName:
       ID ('.' ID)*;

QualifiedNameWithWildcard:
       QualifiedName '.*'?;

Import:
       'import' importedNamespace = QualifiedNameWithWildcard;

ModelElement:
       ( Import | EntityDeclaration | GeneralizationSet ) ';' ;

EntityDeclaration:
       ML2Class | Individual ;

Individual:
       'individual' name=ID
       ':' instantiatedClasses+=[ML2Class|QualifiedName]
                     (',' instantiatedClasses+=[ML2Class|QualifiedName])*
       ( '{' assignments+=FeatureAssignment* '}' )? ;

ML2Class:
       ( SomeFOClass | NonFOClass )
       ('{' ( assignments+=FeatureAssignment | features+=Feature )* '}')? ;

SomeFOClass returns ML2Class:
       ( FOClass | DataType )
       (':' instantiatedClasses+=[ML2Class|QualifiedName]
              (',' instantiatedClasses+=[ML2Class|QualifiedName])*) ?
       ('specializes' superClasses+=[ML2Class|QualifiedName]
              (',' superClasses+=[ML2Class|QualifiedName])*) ? ;

NonFOClass returns ML2Class:
       ( HOClass | OrderlessClass )
       (':' instantiatedClasses+=[ML2Class|QualifiedName]
              (',' instantiatedClasses+=[ML2Class|QualifiedName])*) ?
       ('specializes' superClasses+=[ML2Class|QualifiedName]
              (',' superClasses+=[ML2Class|QualifiedName])*) ?
       ('subordinatedTo' subordinators+=[ML2Class|QualifiedName]
              (',' subordinators+=[ML2Class|QualifiedName])*) ?
       ( categorizationType=CategorizationType
              categorizedClass = [ML2Class|QualifiedName]
              | 'isPowertypeOf' powertypeOf=[ML2Class|QualifiedName] ) ? ;

enum CategorizationType:
       CATEGORIZER | COMPLETE_CATEGORIZER | DISJOINT_CATEGORIZER | PARTITIONER ;

FOClass:
       'class' name=ID ;

DataType:
       'datatype' name=ID ;

HOClass:
       'order' order=INT 'class' name=ID ;

OrderlessClass:
       'orderless' 'class' name=ID ;

GeneralizationSet:
       (isDisjoint?='disjoint'? & isComplete?='complete'?) 'genset' (name=ID)?
       'general' general=[ML2Class|QualifiedName]
       ('categorizer' categorizer=[ML2Class|QualifiedName])?
       'specifics' specifics+=[ML2Class|QualifiedName] (','
              specifics+=[ML2Class|QualifiedName])+ ;

Feature:
       CommonFeature | RegularityFeature ;

CommonFeature returns Feature:
       Attribute | Reference ;

RegularityFeature returns Feature:
       RegularityAttribute | RegularityReference ;

Attribute:
       'att'? name=ID ':'
       ('[' lowerBound=ELEMENTBOUND '..' upperBound=ELEMENTBOUND ']') ?
       (primitiveType=PrimitiveType | _type=[DataType|QualifiedName])
       ('subsets' subsetOf+=[Attribute|QualifiedName]
              (',' subsetOf+=[Attribute|QualifiedName])* ) ? ;

Reference:
       'ref' name=ID ':'
       ('[' lowerBound=ELEMENTBOUND '..' upperBound=ELEMENTBOUND ']')?
_type=[ML2Class|QualifiedName]
       ('subsets' subsetOf+=[Reference|QualifiedName]
              (',' subsetOf+=[Reference|QualifiedName])*) ?
       ('isOppositeTo' oppositeTo=[Reference|QualifiedName])? ;

RegularityAttribute returns Attribute:
       'regularity' 'att'? name=ID ':'
       ('[' lowerBound=ELEMENTBOUND '..' upperBound=ELEMENTBOUND ']') ?
       (primitiveType=PrimitiveType | _type=[DataType|QualifiedName])
       ('subsets' subsetOf+=[Attribute|QualifiedName]
              (',' subsetOf+=[Attribute|QualifiedName])* ) ?
       regularityType=RegularityFeatureType regulatedFeature=[Feature|QualifiedName]
;

RegularityReference returns Reference:
       'regularity' 'ref' name=ID ':'
       ('[' lowerBound=ELEMENTBOUND '..' upperBound=ELEMENTBOUND ']')?
```

```
        _type=[ML2Class|QualifiedName]
         ('subsets' subsetOf+=[Reference|QualifiedName]
         (',' subsetOf+=[Reference|QualifiedName])*) ?
         ('isOppositeTo' oppositeTo=[Reference|QualifiedName])?
         regularityType=RegularityFeatureType regulatedFeature=[Feature|QualifiedName]
;

ELEMENTBOUND returns ecore::EInt:
        '*' | INT ;

enum PrimitiveType returns PrimitiveType:
        STRING | NUMBER | BOOLEAN ;


enum RegularityFeatureType:
        DETERMINES_VALUE | DETERMINES_TYPE | DETERMINES_MIN_VALUE
        | DETERMINES_ALLOWED_VALUES | DETERMINES_ALLOWED_TYPES | DETERMINES_MAX_VALUE
;

FeatureAssignment:
        SingleAttributeAssignment | MultipleAttributeAssignment | ReferenceAssignment
;

SingleAttributeAssignment returns AttributeAssignment:
        'att'? attribute=[Attribute|QualifiedName] '='
        ( individualAssignments+=[Individual|QualifiedName]
                | unnamedIndividualAssignments+=UnnamedIndividual
                | literalAssignments+=Literal ) ;

MultipleAttributeAssignment returns AttributeAssignment:
        'att'? attribute=[Attribute|QualifiedName] '=' '{'
        ((literalAssignments+=Literal (',' literalAssignments+=Literal)*)
                | ((individualAssignments+=[Individual|QualifiedName] |
                unnamedIndividualAssignments+=UnnamedIndividual)
                        (',' (individualAssignments+=[Individual|QualifiedName] |
                        unnamedIndividualAssignments+=UnnamedIndividual))*
                )
        )?  '}';

ReferenceAssignment:
        'ref' reference=[Reference|QualifiedName] '='
        ( assignments+=[EntityDeclaration|QualifiedName]
                | '{' assignments+=[EntityDeclaration|QualifiedName]
                        (',' assignments+=[EntityDeclaration|QualifiedName])* '}' ) ;

Literal:
        ML2String | ML2Number | ML2Boolean ;

ML2String:
        value=STRING;

ML2Number:
        value=NUMBER;

ML2Boolean:
        value=BOOLEAN;

BOOLEAN returns ecore::EBoolean:
        'true' | 'false';

NUMBER returns ecore::EDouble:
        '-'? INT ( '.' INT )? ;

UnnamedIndividual returns Individual:
        {Individual} '[' assignments+=SimpleAttributeAssignment* ']' ;
```

```
SimpleAttributeAssignment returns AttributeAssignment:
       attribute=[Attribute|QualifiedName] '='
       ( individualAssignments+=[Individual|QualifiedName]
              | unnamedIndividualAssignments+=UnnamedIndividual
              | literalAssignments+=Literal
              | '{'
                     ((literalAssignments+=Literal (','
                            literalAssignments+=Literal)*)
                     | ((individualAssignments+=[Individual|QualifiedName] |
                            unnamedIndividualAssignments+=UnnamedIndividual)
                            (',' (individualAssignments+=[Individual|QualifiedName] |
                            unnamedIndividualAssignments+=UnnamedIndividual))*)
              )? '}' ) ;
```

Appendix C. The Bicycle Challenge in ML2

This appendix presents the specification of the Bicycle Challenge, proposed at the workshop MULTI 2017, in terms of ML2.

```
module bicycle.challenge
{
       order 2 class ProductType categorizes Product {
              regularity instancesRegularSalesPrice : Number
                     determinesValue regularSalesPrice
       };
       class Product {
              att regularSalesPrice : Number
              att salesPrice : Number
              att purchasePrice : Number
       };

       class PhysicalObject {
              att weight : Number
              att color : [0..*] Color
       };
       datatype Color { red:Number green:Number blue:Number };

       class ComplexObject specializes PhysicalObject {
              ref components : [1..*] Component
       };
       class Component specializes PhysicalObject;
       class ComplexComponent specializes Component, ComplexObject;

       /* CONSTRAINT: front and rear wheels must have same size */
       /*
        * context Bicycle inv sameSizedWheels:
        *            self.frontWheel.size == self.rearWheel.size
        */
       class Bicycle specializes ComplexObject, Product {
              ref frame : Frame subsets components
              ref fork : Fork subsets components
              ref handleBar : HandleBar subsets components
              ref frontWheel : Wheel subsets components
              ref rearWheel : Wheel subsets components

              att suitableForToughTerrains : Boolean
              att suitableForUrbanAreas : Boolean
              att suitableForRacing : Boolean
       };

       order 2 class FrameType isPowertypeOf Frame;

       class Frame specializes Component, Product {
              att serialNumber : String
       };
       class Fork specializes ComplexComponent, Product {
              ref frontSuspension : [0..1] Suspension subsets components
              ref mudMount : [0..1] MudMount subsets components
       };
       class HandleBar specializes Component, Product;
       class Wheel specializes Component, Product { size : Number };
       class Suspension specializes Component, Product;
       class MudMount specializes Component, Product;

       /* CONSTRAINT: Mountain Bicycles are suited for tough terrains.
        *
        * context MountainBicycle
        *           inv suitableForToughTerrains: self.suitableForToughTerrains
        */
       class MountainBicycle specializes Bicycle {
              ref rearSuspension : [0..1] Suspension subsets components
       };
       /* CONSTRAINT: City Bicycles are suited for urban areas
        *
        * context CityBicycle
        *           inv isSuitedForUrbanAreas: self.suitableForUrbanAreas
        */
       class CityBicycle specializes Bicycle;
       /* CONSTRAINT: Racing bicycle are suited for racing and urban areas,
        *           but are not suited for tough terrains.
        *
        * context RacingBicycle
        *           inv suitableForRacing: self.suitableForRacing
        *           inv isSuitedForUrbanAreas: self.suitableForUrbanAreas
        *           inv suitableForToughTerrains: not self.suitableForToughTerrains
        */
       class RacingBicycle specializes Bicycle {
              att isCertified : Boolean
       };

       /* CONSTRAINT: Race forks cannot have suspensions nor mud mounts */
       /*
        * context RacingFork
        *           inv noSuspensions: self.frontSuspension->isEmpty()
        *           inv noMudMount: self.mudMount->isEmpty()
        */
       class RacingFork specializes Fork;
       class RacingFrame specializes Frame {
              att topTubeLength : Number
              att downTubeLength : Number
              att seatTubeLength : Number
       };

       class SteelFrame specializes Frame;
       class AluminumFrame specializes Frame;
       /* CONSTRAINT: carbon frames ask for carbon or aluminum wheels only */
       /*
        * context Bicycle inv carbonFrameConstraint:
        *           self.frame.oclIsKindOf(CarbonFrame) implies
        *           (self.frontWheel.oclIsKindOf(AluminumWheel)
        *                  or self.frontWheel.oclIsKindOf(CarbonWheel))
        *           and (self.rearWheel.oclIsKindOf(AluminumWheel)
        *                  or self.rearWheel.oclIsKindOf(CarbonWheel))
        */
       class CarbonFrame specializes Frame;

       disjoint genset
              general Frame
              specifics SteelFrame, AluminumFrame, CarbonFrame;

       order 2 class RacingBicycleType categorizes RacingBicycle {
              regularity minimumWeight : Number determinesMinValue weight
              regularity ref allowedFrameTypes : [0..*] FrameType
                     determinesAllowedTypes frame
       };

       /* CONSTRAINT: Mandatory certification */
       /*
```

```
         * context ProRacingBicycle inv proRacingBicycleRequeriments:
         *             self.isCertified
         */
        class ProRacingBicycle :RacingBicycleType specializes RacingBicycle {
                att minimumWeight  = 5.200
                ref allowedFrameTypes =  {AluminumFrame, CarbonFrame}
        };

        class AluminumWheel specializes Wheel;
        class CarbonWheel specializes Wheel;

        class ChallengerA2XL :RacingBicycleType, ProductType
                specializes ProRacingBicycle
        {
                att instancesRegularSalesPrice = 4999.00
                ref frame : RocketA1XL subsets frame
        };

        order 2 class PhysicalObjectType isPowertypeOf PhysicalObject {
                att instancesWeight : [0..1] Number
        };

        class ProRacingFrame specializes RacingFrame;
        class RocketA1XL :ProductType specializes ProRacingFrame {
                att instancesWeight = 0.920
        };

}
```

Appendix D. UFO in ML2

This appendix presents the specification of UFO accounting for the multi-level aspects of the ontology. The specification is captured in the module “ufo” as presented below.

```
module ufo {

	class Endurant;
	class Substantial specializes Endurant;
	class Moment specializes Endurant {
		ref inheresIn : Endurant
	};
	disjoint complete genset existential_dependece
		general Endurant
		specifics Substantial, Moment;

	class Relator specializes Moment;
	class IntrinsicMoment specializes Moment;
	disjoint complete genset unique_existential_dependence
		general Moment
		specifics IntrinsicMoment, Relator;

	class Quality specializes IntrinsicMoment;
	class Mode specializes IntrinsicMoment;
	disjoint complete genset
		general IntrinsicMoment
		specifics Quality, Mode;

	class ExternallyDependentMode specializes Mode {
		ref partOf : Relator
		ref externallyDepedentOn : [1..*] Endurant
	};

	order 2 class EndurantUniversal categorizes Endurant;
	order 2 class SubstantialUniversal specializes EndurantUniversal
		categorizes Substantial;
	order 2 class MomentUniversal specializes EndurantUniversal
		categorizes Moment;
	disjoint complete genset existential_dependece_of_instances
		general EndurantUniversal
		specifics SubstantialUniversal, MomentUniversal;

	order 2 class RelatorUniversal specializes MomentUniversal
		categorizes Relator;
	order 2 class IntrinsicMomentUniversal specializes MomentUniversal
		categorizes IntrinsicMoment;
	disjoint complete genset unique_existential_dependence_of_instances
		general MomentUniversal
		specifics IntrinsicMomentUniversal, RelatorUniversal;

	order 2 class QualityUniversal specializes IntrinsicMomentUniversal
		categorizes Quality;
	order 2 class ModeUniversal specializes IntrinsicMomentUniversal
		categorizes Mode;
	disjoint complete genset
		general IntrinsicMomentUniversal
```

```
              specifics QualityUniversal, ModeUniversal;

       order 2 class MixinUniversal specializes SubstantialUniversal;
       order 2 class SortalUniversal specializes SubstantialUniversal;
       disjoint complete genset
              general SubstantialUniversal
              specifics SortalUniversal, MixinUniversal;

       order 2 class RigidMixin specializes MixinUniversal;
       order 2 class AntiRigidMixin specializes MixinUniversal;
       disjoint complete genset
              general MixinUniversal
              specifics RigidMixin, AntiRigidMixin;

       order 2 class Category specializes RigidMixin;
       order 2 class PhaseMixin specializes AntiRigidMixin;
       order 2 class RoleMixin specializes AntiRigidMixin;
       disjoint complete genset
              general AntiRigidMixin
              specifics PhaseMixin, RoleMixin;

       order 2 class RigidSortal specializes SortalUniversal;
       order 2 class AntiRigidSortal specializes SortalUniversal;
       disjoint complete genset
              general SortalUniversal
              specifics RigidSortal, AntiRigidSortal;

       order 2 class Kind specializes RigidSortal partitions Substantial;
       order 2 class Subkind specializes RigidSortal subordinatedTo Kind;
       disjoint complete genset
              general RigidSortal
              specifics Kind, Subkind;

       order 2 class Phase specializes AntiRigidSortal subordinatedTo Kind;
       order 2 class Role specializes AntiRigidSortal subordinatedTo Kind;
       disjoint complete genset
              general AntiRigidSortal
              specifics Phase, Role;

}
```