University of Mannheim

Department of Business Informatics and Mathematics
Chair of Software Engineering - Prof. Dr. Colin Atkinson

Master Thesis at the University of Mannheim in Business Informatics

Supervisor: Ralph Gerbig

Specification and Implementation of a Deep OCL Dialect

Dominik Kantner
<dkantner@mail.uni-mannheim.de>

Mannheim, September 13, 2014

Abstract

The deep modeling tool Multi-level Modeling and Ontology Engineering Environment (Melanee) developed by the Software Engineering Group of the University of Mannheim allows clean and strict meta-modeling across multiple classification levels within Eclipse. Modeling with Melanee comprises two dimensions, a linguistic and an ontological dimension. As Melanee is fully embedded in the Eclipse Modeling Framework, it is usable with the Object Constraint Language (OCL).

However, the current OCL implementation is not aware of multi-level modeling features like the distinction between ontological and linguistic classification and the existence of multiple ontological levels. The aim of this thesis is to extend the current OCL implementation so that it is multi-level aware. First, a deep OCL dialect is elaborated which takes deep modeling features into consideration. Based on this dialect, an implementation of an interactive level-agnostic OCL Console will enable queries of different values of different model elements.

Contents

Glossary

CIM Computational Independent Model

CWM Common Warehouse Model

DSL Domain Specific Language

EMF Eclipse Modeling Framework

EMOF Essential MOF

GMF Graphical Modeling Framework

IDE Integrated Development Environment

LML Level-Agnostic Modeling Language

MBD Model-Based Development

MDA Model Driven Architecture

MDD Model-Driven Development

MDE Model-Driven Engineering

MDSD Model-Driven Software Development

MOF Meta Object Facility

OCA Orthogonal Classification Architecture

OCL Object Constraint Language

OMG Object Management Group

PIM Platform Independent Model

PLM Pan-Level Model

PSM Platform Specific Model

QVT Query View Transformation

TMF Textual Modeling Framework

UML Unified Modeling Language

XMI XML Metadata Interchange

List of Figures

Listings

1. Introduction

The software engineering methodology of Model-Driven Development (MDD) is a well established development approach and highly appreciated among software developers and researchers. MDD focuses on models which have a major impact on the quality of software. One of the main pillars in MDD is the Unified Modeling Language (UML) [29]. In the past, however, fundamental weaknesses of the UML have been revealed like the limitations of considering only one form of classification [9] and supporting only two ontological levels when modeling a problem domain ([7], [24]). As a consequence, new frameworks and tools have been developed in order to address these weaknesses.

The Level-agnostic Modeling Language (LML), specified by Atkinson et al. [6], is a concrete definition of a language which also was motivated by the weaknesses of the UML and designed to support deep modeling. Despite its weaknesses and limitations, the modeling features and the concrete syntax of the UML are de facto standard so that LML's concrete syntax is adhering to its paradigms. In order to give practitioners the possibility to create LML models in an interactive way, a LML model editor based on the Orthogonal Classification Architecture (OCA) was developed by the Chair of Software Engineering at the University of Mannheim. This editor is referred to as the Multi-level Modeling and Ontology Engineering Environment (Melanee) [5]. Melanee was steadily extended in the past so that it currently consists of several other software components besides the ability of creating LML models, for example a multi-level aware transformation support. Moreover, LML models can be queried in the Object Constraint Language (OCL) [31] by a built-in OCL console.

In general, OCL enables queries on model elements as well on its properties and the definition of constraints. Regarding this, the capabilities of OCL can also add additional value when creating and managing LML models in Melanee. Nevertheless, the current OCL implementation is not multi-level aware which means that the distinction between ontological and linguistic dimension as well as the support of multiple ontological levels is not taken into consideration.

The aim of this thesis is to extend the existing OCL implementation in Melanee so that it is multi-level aware. A foundation of this implementation will be laid by the specification of a deep OCL dialect which takes multi-level features into account. This dialect is realized by the implementation of a level-agnostic OCL console which allows queries of different values of different model elements in a LML diagram.

The first part of this thesis describes the foundations of the work. They are necessary to have a basic knowledge in the ensuing chapters. In the next chapter, the specification of a deep OCL dialect is elaborated. Requirements for the evaluation of OCL expressions and the definition and validation of constraints are examined and proposals for these

requirements are given. In chapter 4, the core concepts of the implementation of the level-agnostic console and its underlying OCL service are explained. Furthermore, the current status of the deep OCL validation service is shown. Chapter 5 presents the evaluation of the level-agnostic OCL console. Referring to the Royal and Loyal example introduced by Warmer and Kleppe [39], OCL expressions are entered in the implemented console and their results are evaluated. In the next chapter, proposals for future work are made and related work in research is presented. Finally, the thesis closes with a conclusion.

2. Foundations

In this chapter the foundations are described which build the basis of this thesis. At first, the concept of Model-Driven Development is introduced. After this, the principles of deep modeling are explained. Then, the Object Constraint Language is presented. Next, Eclipse plug-in development is described. Hereafter, the Eclipse Modeling Framework and some of its components which are relevant with respect to this thesis are presented. Finally, Melanee which is the tool upon which the work of this thesis is built is introduced.

2.1. Model-Driven Development

This section deals with the software engineering methodology *Model-Driven Development* (MDD). At first motivation and definitions of this methodology are described. After this the most important standardization approach concerning MDD the *Model Driven Architecture* (MDA) is presented. Finally the concept of *meta-modeling* and its meaning in MDD are illustrated.

2.1.1. Motivation and Definition

According to France and Rumpe [22], the most important factor in developing software is the gap between problem domain and implementation domain. They state that such a "gap exists when a developer implements software solutions using abstractions that are at a lower level than those used to express the actual problem" [22]. This behavior can invoke significant accidental complexities meaning that the developer uses implementation specific workarounds to map the problem domain onto the systems implementation technology so that the actual problem gets blurred.

Because of the aforementioned gap, many software engineering researchers are focusing their work on the importance of models. Selic [36] claims that "models help us understand a complex problem and its potential solutions through abstraction" and that models "are much closer to the problem domain relative to most popular programming languages" [36]. As reported by France and Rumpe [22], models help to depict a human understandable form of a system or also to show information so that it is interpretable for machines. Furthermore "the potential benefits of using models are significantly greater in software than in any other engineering discipline" [36].

With respect to the importance of models, the term of *Model-Driven Development* has emerged. Stahl et. al [37] refer to *Model-Driven Software Development* (MDSD) instead of MDD, France and Rumpe [22] are regarding MDD as synonymous to *Model-Driven engineering* (MDE). In the following, however, the term MDD will be used.

Selic [36] states that "MDD's defining characteristic is that software development's primary focus and products are models rather than computer programs" [36]. Referring to

Atkinson and Kühne [3], the MDD approach does not demand the complete implementation specific knowledge from the developers but let them model what the functionalities are and what architecture the system should have. This is in accordance to France and Rumpe [22] which argue that a major goal of MDD research "is to produce technologies that shield software developers from the complexities of the underlying implementation platform" [22] so that developers model the problem domain on a more abstract level. Stahl et. al [37] highlight the distinction between *Model-Based Development* (MBD) and MDD. For them models are not only necessary for documentation, they are also a part of the software meaning that the software is driven by models. So MDD is enabling to model on a abstract but comprehensible level which then can be "systematically transformed to concrete implementations" [22] which is the key idea to close the gap between problem domain and implementation domain.

The automation of the transformation of the model towards a concrete program is a main pillar in MDD research. For example, Selic [36] claims that "a key premise behind MDD is that programs are automatically generated from their corresponding models" [36]. According to Atkinson and Kühne [3], one goal among others of MDD is to automate complex but routine programming tasks which result in increased development speed and higher software quality.

Another aspect in MDD research is standardization "as it provides a significant impetus for further progress because it codifies best practices, enables and encourages reuse, and facilitates interworking between complementary tools" [36]. The most relevant standardization initiative focusing on MDD is the *Model-Driven Architecture* (MDA) which is presented in the next section.

2.1.2. The Model-Driven Architecture

The *Object Management Group* (OMG) [26] announced the *Model-Driven Architecture* [27] which supports MDD or can be regarded as a framework for MDD [22]. The OMG is an open-membership concortium working on establishing standards for the computer industry which was founded in 1989.

Essentially, the MDA has five main goals respectively motivations [37]. First of all, the *interoperability* of software systems so that they are independent of their manufacturers. Another aspect is the *platform independence* of software systems. Furthermore the provision of guidelines and standards for defining system functionality specifications aims to establish an improved *maintainability* of software artifacts. This is achieved by a *separation of concerns* (on the one hand functionality specifications, on the other hand system specifications) and by an improved *manageability of technological changes*. In the following some core concepts of the MDA are described roughly.

The aforementioned separation of concerns results in basically three kinds of models which are arranged in a consecutive way. The most abstract model is the *Computation Independent Model* (CIM) which represents the domain of the model without regarding computational complexities. The CIM provides the basis for the *Platform Independent Model* (PIM) which describes the architecture and the behavior of the system platform independently. The subsequent model is the *Platform Specific Model* (PSM) which contains the required information concerning platform specific technologies.

The *Meta Object Facility* (MOF) [30] constitutes the core of the MDA. The MDA uses the MOF as its meta meta-model and thus serves as formal basis for the definition of meta-models (see 2.1.3). Moreover, it is the industry standard-environment for transporting models from one application to another, storing them in and retrieving them from repositories, rendering them in different formats, e.g. in the OMG's XML based standard format for model transformation *XML Metadata Interchange* (XMI), and finally for transforming as well as generating code from them. Several implementations of the MOF have been developed which can be considered as subsets of the MOF. The probably most known example is the *Essential MOF* (EMOF) which evolved as a consequence of the influence of the ECORE meta meta-model of the Eclipse Modeling Framework (EMF) introduced in chapter 2.5.1.

The *Unified Modeling Language* (UML) [29] is the OMG's most used specification and provides, besides the MOF, the key foundation of the MDA. The aim of the UML "is to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes" [29]. Basically, the UML is perceived as an instance of the MOF and thus the MOF forms the UML meta-model (see 2.1.3).

Another concept established in the MDA is the *Query View Transformation* (QVT) specification [28], which forms a standardization of how models are transformed into models, i.e. so called model-to-model transformations. QVT itself is a part of the MOF and enables the transformation of MOF-based models.

Moreover, there are further technologies besides the *Object Constraint Language* (OCL) (see detailed description in 2.3) belonging to the MDA which are not mentioned here because they are not important in the context of this thesis.

2.1.3. Meta-Modeling

With respect to Stahl et al. [37], *meta-modeling* is one of the most important aspects of MDD. Meta-models are models of models meaning that they describe the formal structure of a model. This refers to the so called abstract syntax. In contrast to this, the concrete syntax defines how the model should be represented visually. Meta-modeling knowledge helps to meet MDD challenges [37] which are roughly explained in the following.

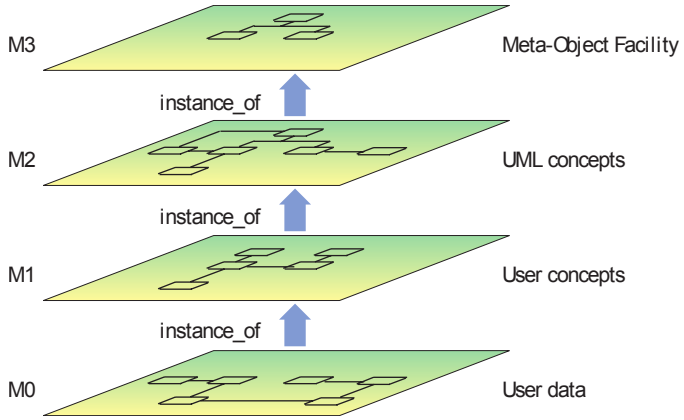

Figure 1: The OMG's four layer infrastructure adapted from [3].

Meta-modeling helps in the construction of *Domain Specific Languages* (DSL) as it scribes their abstract syntax. According to France and Rumpe [22] a DSL "consists of constructs that capture phenomena in the domain it describes" [22] and can help to close the gap between the problem domain and the corresponding implementation. Additionally, meta-modeling enables model validation through constraints defined in meta-models which then can be checked. Another challenge supported by meta-modeling are model-to-model transformations which can be defined by rules in the meta-model. Finally, code generation can be realized by generation templates specified in the meta-model.

As UML is used today for modeling in many cases, it is of practical relevance to clarify the concept of meta-modeling in the context of the UML. The UML is part of the OMG's modeling infrastructure which consists of four layers respectively meta-levels presented in Figure 1.

Layer M3 is the OMG's meta meta-model also referred as MOF. The MOF serves to specify modeling languages on M2 for which UML is a famous example. However, UML is only one modeling language out of many possible modeling language approaches. So the key idea and advantage of this infrastructure is that other domain-specific languages and standardized modeling languages compatible to the MOF can be easily defined ([37], [3]). Consequently, the interchange between MOF-based models is supported. Another example for a modeling standard besides the UML on the M2 is the *Common Warehouse Metamodel* (CWM). Elements of the M1 model are instances of elements in the M2 (e.g. the construct *Class* of the UML). Atkinson and Kühne [3] refer to user concepts concerning the M1. They state that the user concepts on the M1 hold the model for the concrete user data which reside at the M0.

Stahl et al. claim [37] that, despite the fact that UML can be simply regarded as instance of the MOF, various details have to be considered which are roughly discussed next. First of all UML was designed before the MOF and in the beginning the UML was not formally defined. Because of this, the MOF was created to define the UML formally.

Thus the initial artifact was the UML, not the MOF. Furthermore another potential for misunderstandings originates from the fact that the MOF uses the concrete syntax of the UML, the language it formally models. Finally, there are some elements which exist in both the UML and the MOF. Although these elements have the same name, they are not identical.

2.2. Deep Modeling

In this section the basis for the approach of *deep modeling* is established. At first, the underlying motivation and the concept of deep modeling is introduced. After this, the *Level-Agnostic Modeling Language* (LML) is presented which is a modeling language following the paradigm of deep modeling.

2.2.1. Motivation and Concept

Several researches ([8], [13], [24]) revealed profound limitations in meta-modeling adhering to the four layer infrastructure, which was mentioned in the previous section. The fundamental problem of this infrastructure is that only one kind of instance-of relationship is considered and that it is restricted to only two meta-levels at the same time. According to De Lara et al. [13], two meta levels are enough to cover the *lingusitic* case meaning the abstract syntax of a model, but not to address the *ontological* case where the instance-of relationship refers to a logical domain.

As a result of this dilemma, modelers are forced to squeeze models into two levels which would naturally encompass multiple levels. Thus, they have to build workarounds to model the problem domain. An example for such a workaround is depicted in Figure 2. In the example there are two domain specific kinds of instance-of relationships. SportsCar is an instance of a CarType, whereby MyPorsche is an instance of SportsCar. Despite the fact that there are three meta-levels in this example, the modeler has to manage the problem domain with only two levels which leads to more complex models and also creates an overhead for developers. This is referred to *accidental complexity* which is complexity not induced by the complexity of the problem domain but caused by the complexity of the solution approach [33]. For example, the users of the model have to deal with two kinds of instance-of relationships: The is-of-type relationships like the relationship between Car and CarType and the instance-of relationship between language types and language instances like the relationship between SportsCar and CarType. The is-of-type relationships has to be manually maintained by the developers as there is no language support for typechecking the domain instances regarding their domain types. Furthermore the element Car is an artificial pattern which does not represent any element in the original

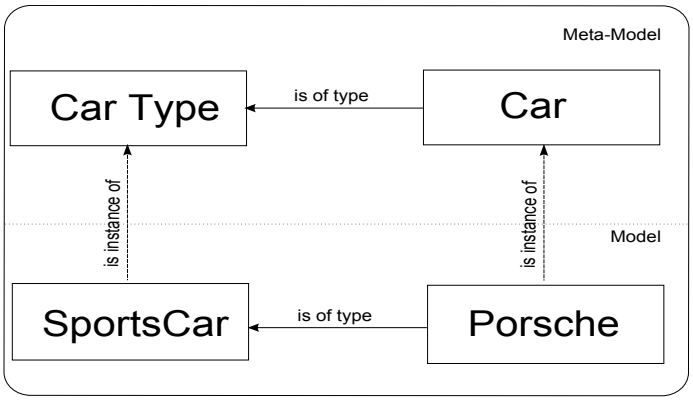

Figure 2: Example of squeezing three domain levels into two.

domain. It is compulsory in the two level model and its objective is to serve as class for the domain instance MyPorsche.

In order to overcome these weaknesses, the concept of *multi-level* or *deep modeling* arose in MDD research. This thesis will refer to the term of deep modeling. Basically, the concept of deep modeling is based on two main ideas. The first one encompasses the principle of dual instantiation ([9], [13]). This means that an element can be both, a linguistic instance and an ontological instance. The linguistic type defines the structure and the presentation of an element whereas the ontological type refers to some logical domain. The second idea is to increase the flexibility of meta-modeling by allowing an arbitrary number of levels regarding a specific domain. In this context, the concept of *Deep Characterization* comes into play. Deep Characterization allows elements of a certain level in a model not only to control the characteristics of elements of the level below but also to control the characteristics of elements of all other further levels below. Gonzalez et al. [24] suggest to achieve Deep Characterization by using *Powertypes* which is a mechanism based on the UML. In contrast to this, Atkinson and Kühne [9] are referring to a mechanism called *Deep Instantiation* where the concept is "to assign a potency value to model elements and their fields, indicating how many times they can be instantiated" [9].

2.2.2. The Level-Agnostic Modeling Language

A concrete definition of a language designed with the primary intention of enabling deep modeling is the *Level-Agnostic Modeling Language* (LML) [6]. With respect to Atkinson et al. [6], it has four major goals. First of all it should be UML-like. Albeit the UML has fundamental weaknesses like not really supporting the concept of ontological engineering, its modeling features and its concrete syntax are de facto standard and so it is of practical relevance to use them. Another goal is to be level-agnostic which basically means that LML should support deep modeling. The third goal demands to correspond to all

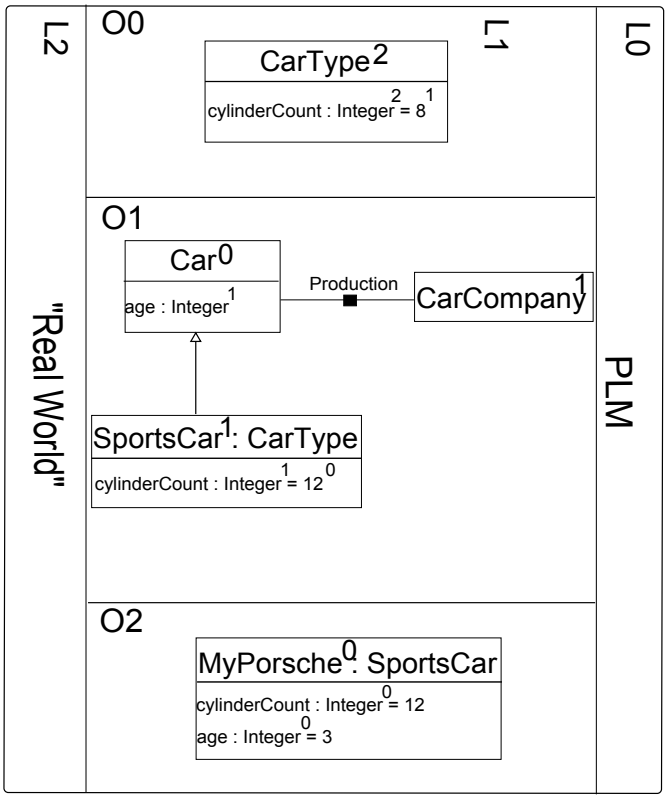

Figure 3: LML diagram embedded in the OCA.

mainstream modeling paradigms besides the UML. The last goal of the LML is to provide a unified set of reasoning and checking services.

In the following, the core concepts of the LML are described with the help of an example in Figure 3 which shows a LML Diagram embedded in the *Orthogonal Classification Architecture* (OCA). The OCA builds the framework for LML. In contrast to the 4 level modeling infrastructure of the OMG, the OCA consists of two classification dimensions which are orthogonal to each other. Each dimension can be divided into a number of levels in accordance with the principle of *strict metamodeling*. In respect of Atkinson et al. [2], adhering to strict meta-modeling means that "instanceOf (classification) relationships can cross model-level boundaries, and that every element in one level must be a direct instance of exactly one element of the level above" [2].

There are three levels in the linguistic dimension (L0,L1,L2) and an arbitrary number of levels in the ontological dimension. The LML resides in the linguistic layer L1. Every element of L1, independently from its ontological level, is an instance of an element in the L0. The L0 consists of the *Pan-Level Model* (PLM) which can be regarded as the linguistic meta-model and thus the abstract syntax of the LML. The L2 is perceived, similar to the bottom level in the four layer infrastructure of the OMG, as the real world. From an ontological point of view, each element is an instance of an element defined in

the level above (except the elements in O0). For instance, SportsCar is an ontological instance of CarType.

When there are more than 2 ontological levels, elements in the middle (i.e. elements neither in O0, neither in the highest ontological level) can have both, a type respectively class and an object facet. For example, SportsCar is an instance of a CarType but serves also as type for MyPorsche. In order to address this dual facet, a unified concept was introduced by Atkinson [4]. This concept is called *clabject* which combines the two terms class and object. The meta-type characteristics of a clabject are modeled and defined in the PLM. Here the clabject is modeled as supertype of an *entity* and a *connection*. Thus, a clabject can be both, either an entity or a connection. According to Atkinson et al. [6], "the former play a role similar to classes/objects in the UML and the latter play a role similar to associations/links". Examples for an entity in Figure 3 are the entities CarType, Car and SportsCar. An example for a connection is the Production connection, which in this case is displayed in the so called "dotted" format. Another format is the exploded format which enables to show more information like attributes of the connection. Because connections are clabjects and can also have attributes, they are comparable to association classes of the UML.

Another artifact in the LML is the notion of *potency* which allows a precise definition of instance-of relationships over multiple ontological levels and therefore enables the aforementioned mechanism of deep instantiation. A potency takes non-negative integer values. An instantiation lowers the potency by one which means that the potency of an instance of an ontological type is one lower than the potency of its ontological type. There are three kinds of potency which are the potency of a clabject, the potency of a feature and the potency of an attribute value.

The potency of a clabject specifies over how many levels it can be instantiated. The CarType in Figure 3 has a potency of two which means that it can have instances over two more levels (i.e. SportsCar in O1 and MyPorsche in O2). A potency of 0 for a clabject, like this is the case for a Car, can be regarded as abstract class as it cannot be instantiated. The potency of a feature is also called *durability*. The durability of a feature defines over how many levels this feature is expected to be present in the instances of the clabject where it is specified. For instance, the attribute cylinderAmount of the CarType has a potency of two so that is present in the instances of CarType one and two levels further below. The potency of an attribute value is referred to as the *mutability* of an attribute. It specifies over how many levels the value may be changed from the default. For example in CarType the value of cylinderAmount has a potency of one, but the Attribute CylinderAmount itself has a potency of two. So in general an attribute's mutability can be smaller than its durability, but it must be not greater. In this case, the cylinderAmount attribute is present over three levels, but it can only be changed in

CarType (resided in O0) and SportsCar (resided in O1). It is not possible to change the value of cylinderAmount in MyPorsche resided in O2.

2.3. The Object Constraint Language

This chapter gives an introduction in the *Object Constraint Language* (OCL) [31] that helps to establish an understanding in the fundamentals of this language. This comprehension is of particular importance to grasp the ideas of this thesis. The section is structured as follows. At first, the OCL is defined and its main characteristics are outlined. After this, the OCL meta-model is presented. Finally, some OCL features are described.

2.3.1. Definition and Characteristics

The Object Constraint Language (OCL) is the MOF-compliant standard for specifying expressions that enrich models in software development with additional information. According to Stahl et al. [37], OCL is independent of modeling language so that OCL can be used at various meta-levels. With respect to the OMG's four layer infrastructure, OCL can be used on one hand to refine (meta-)models at M2 for specifying modeling languages, on the other hand modelers can also use the capabilities of the OCL like in a concrete UML model, which is resided at M1.

Warmer et al. [39] are referring to four major characteristics of the OCL which are listed and explained next. At first, OCL is *a query and a constraint language*. For them "a constraint is a restriction on one or more values of (part of) an object-oriented model or system" [39]. Thus, constraints set conditions for values of models which have to be fulfilled so that these values are valid in the modeled system. In Listing 1, for example, a car is restricted to have a maximum age of 20 (years).

```
context Car
inv: self.age <= 20
```

Listing 1: Example of a constraint in OCL.

Constraints which are placed on elements on the M_n affect elements on the M_{n-1}. Therefore, when creating an instance of a car, this instance is only valid if its age is not greater than 20 years. Constraints are built upon *OCL expressions*. Every expression represents respectively returns a value which can be a primitive type like a Boolean or Integer but also an object or a collection of references to objects. In general, one can state that every expression of type Boolean can be used as a constraint. Furthermore, OCL expressions are also used when querying values of a specific model element. For example, the query "self.age" on a car would return the age of the car.

The second characteristic is that albeit OCL has a *mathematical foundation*, it uses *no mathematical symbols*. OCL is founded on mathematical theory and predicate logic. This mathematical precision is of special importance in the context of the MDA where models need to be transformed. Through the mathematical foundation models are getting more suitable for automated tools performing model transformation, checking consistency and generating code. However, a mathematical notation might be not appropriate for a standard language which is used widely among practitioners with very different backgrounds. For some of these practitioners a mathematical notation would be to complex to understand. Therefore, OCL became a well balanced language that consists of a modeling language precision of mathematics, but also promises the ease of use of a natural language. In essence, OCL is precise and unambiguous which can be easily read and written by all people, also those who are not mathematicians or computer scientists.

The third characteristic holds that OCL is a strongly *typed language*. Most of the OCL expressions will be written before an executable form of a model exists, so it has to be possible to check an expression without having to generate an executable version of a model. As a typed language, this is attainable with OCL so that errors in models also can be detected at an early stage (e.g. an Integer cannot be compared to a String).

Finally, OCL is a *declarative language* meaning that it simply states *what* a constraint or an expression should do but not *how* (like this is the case in procedural languages). In order to achieve this, an OCL expression has no side effects. Thus, OCL expressions can be made at a high level of abstraction without going into implementation details of this expression. Nevertheless, OCL remains very precise. Hence, OCL is perceived as an essential artifact in the domain of platform-independent modeling.

To sum up, the combination of easy to grasp visualized models (consisting of elements like boxes and arrows) with the previously listed advantages of OCL allows practitioners to create models, which are easy to understand, but also formal, precise and consistent.

2.3.2. The OCL Meta-Model

With respect to Cadavid et al. [11] the MOF and the OCL are commonly used for meta-modeling. The MOF is utilized to specify a domain model and the OCL to add well-formedness rules to the domain. In accordance to this, the MOF core specification [30] as well as the UML infrastructure [29] which share a common core were developed in parallel with the OCL specification [31]. The OCL specification "contains a well-defined and named subset of OCL that is defined purely based on the common core of UML and MOF" [31]. In context of this thesis, however, it is sufficient to describe the meta-model of the *EssentialOCL* which is required to work with the EMOF, the meta-model on which Melanee (see section 2.6) is based on. According to the OCL Specification [31], "EssentialOCL is the package exposing the minimal OCL required to work with EMOF"

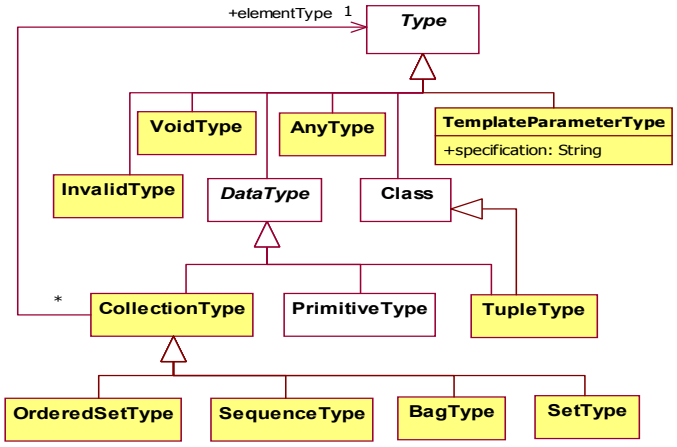

Figure 4: The Types package as specified in [31].

[31]. In the following, the meta-model of EssentialOCL as well as its connections to EMOF are briefly described.

The EssentialOCL depends on the EMOF package as it explicitly references classes from EMOF. It consists of two packages: the *Types* package, describing the type system in OCL and the *Expressions* package, describing OCL expressions.

The Types Package

OCL is a typed language so that each expression has to conform to a type. Therefore a meta-model for the type concept is provided in the OCL specification [31], which is shown in Figure 4.

The types imported from the EMOF package are visualized as boxes with a white background color. As opposed to this, the colored boxes represent types that are specified especially for the EssentialOCL. All elements are derived from the type *Type* which is imported from the EMOF. This enables type compatibility between OCL and EMOF. Additional classes adapted from the EMOF package are the classes *DataType*, *Class* and *PrimitiveType*, for instance Integer or Boolean. Examples for types which are defined in the OCL context are *AnyType* and *CollectionType*. AnyType serves as metaclass for the type OCLAny which is the supertype of all other types so that all properties of OCLAny are available for each element in a model (e.g. the operations oclIsTypeOf and oclIsKindOf). The type CollectionType describes a type storing a collection of elements of a specific type. There are four collection types. A *Set* is a collection which has no duplicate elements. A special kind of a Set is the *OrderedSet* where the elements are ordered. A *Bag* is a collection which also contains duplicate elements. If the elements in a Bag are ordered, it is called a *Sequence*.

The Expressions Package

The OCL specification [31] includes also an expression package which is displayed in Figure 5. It allows enriching models built upon the MOF with expressions and constraints.

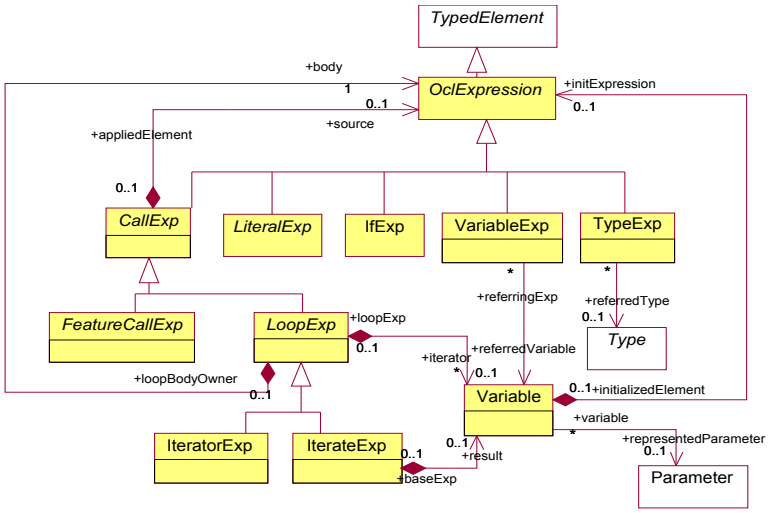

Figure 5: The Expressions package as specified in [31].

Classes adopted from the EMOF package are visualized again by a white background color. In the following the most important artifacts of this package are briefly described.

The basic class is *OCLExpression*. An OCLExpression always conforms to a type compatible to the EMOF because it is derived from the class *TypedElement* which holds a reference to the already described class Type of the EMOF package. Expressions with a Boolean type can be used to build constraints, those of AnyType can be used for multiple purposes (e.g. defining queries, initial attribute values etc.).

A *CallExp* permits practitioners creating composite OCLExpressions. It refers to a feature or a predefined iterator for a collection. A feature can be either an operation or a property which also can consist of a navigation to an AssociationClass. Each CallExp has exactly one source identified by an OCLExpression. Furthermore, the result value of a CallExp is the evaluation of the corresponding feature or iterator.

The *VariableExp* forms the basis for the declaration of variables in the OCL environment. This expression references a *Variable* which is a TypedElement for passing data in expressions. One example for a Variable is *self* which holds the information for the contextual instance an OCL expression can be specified for. The *LiteralExp* is basically an expression producing a value which is identical with its expression symbol. Examples are an Integer like 1 or a String like "this is an LiteralExp". An *IfExp* is an expression which results in one of two alternative expressions depending on the evaluated value of a condition. A *TypeExp* enables to refer to an existing type which is used in particular when invoking the operations *oclIsKindOf*, *oclIsTypeOf* and *oclAsType*.

Connection between EMOF and the OCL Meta-Model

Figure 6 visualizes the connection between EMOF and the OCL meta-model. It is based on the connection between the MOF and OCL described by Cadavid et al. [11] and adapted to the EMOF context. The most important concept here is the notion of *Expres-*

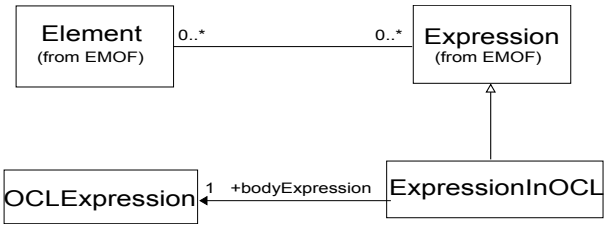

Figure 6: The Connection between EMOF and OCL.

sionInOCL that binds an element coming from the EMOF core to an OCLExpression. It is the entry point or the top-level container class representing the top of a abstract syntax tree in which OCLExpression is defined recursively. Hence, the illustrated connection between EMOF and OCL is the foundation to define nested OCLExpressions on elements in models which are based on the EMOF.

2.3.3. OCL Features

This part describes a snippet of the OCL features which are listed in the OCL Quick reference [10]. Their description is also based on the "Reference Manual" part of the work by Warmer et al. [39] and the OCL Specification [31]. Those features which are perceived as most relevant with regard to the subject of this thesis are selected. They are described with the help of examples written in OCL. First some *OCL constructs* are explained. Building on this, *OCL expressions* and the *OCL Standard Library* with its predefined types and operations are presented.

OCL Constructs

A very import construct is built with the help of the keyword *context* which specifies the model entity, i.e. the context, for an OCL expression. A context can be a model type (i.e. type, component, class etc.), an attribute or an operation. Listing 1, for example, sets the context for an expression to the type Car. Depending on the context, expressions can have different functions, which also will be shown in this section. The so called *contextual type* of an expression refers to the type of the object for which the expression will be evaluated. In Listing 1, the contextual type is equal to the context which is Car. In case the context is an attribute or an operation, the contextual type is equal to the type for which the attribute respectively operation is defined. As OCL expressions are evaluated on the instance of the contextual type, it is necessary to distinguish between the contextual type on the one hand and the *contextual instance* on the other hand.

With regard to the context, different OCL constructs respectively constraints can be applied. When the context is a type, the two possible constructs are *invariants* and the *definition of new attributes or operations* for this type. Invariants (keyword *inv*) can be associated with a type and must be met by all instances of this type, so it acts as a

constraint for a specific model type. Regarding Listing 1 all instances of Sportcar must be not older than 20. An OCL expression that expresses an invariant is always of the type Boolean. Defining attributes or operations (keyword *def*) on a type with an OCL expression implies that the attributes or operations have to be present in the instance of the contextual type. Listing 2 shows a definition of a Boolean attribute hasAcceptableAge which evaluates to true if the Car instance is not older than 20.

```
context Car
def: hasAcceptableAge : Boolean = age <= 20
```

Listing 2: Example of the *definition* construct in OCL.

When the context is an attribute, OCL expressions can be used to specify *derivation rules* or *inital values* for attributes. A derivation rule (keyword *derive*) defines the value of an attribute which has to evaluate to the value of the corresponding OCL expression. Listing 3 shows an example where the value of the Integer attribute licensedYearsLeft is derived by the OCL expression which calculates the difference between 20 and the age of a Car instance.

```
context Car::licensedYearsLeft:Integer
derive: 20 - age
```

Listing 3: Example of the *derive* construct in OCL.

An initial value (keyword *init*) can be specified in order to assign an attribute a value at the moment a contextual instance of a type is created. For example, in Listing 4 the initial value of the age of a Car instance is meant to be zero.

```
context Car::age:Integer
init: 0
```

Listing 4: Example of the *init* construct in OCL.

In case the context is an operation, OCL expressions can be used either as *pre- and postconditions* or as *body of query operations*. Pre- and postconditions (keywords *pre* and *post*) state conditions which have to be true when an operation starts respectively ends its execution and thus serve as constraints on operations. In Listing 5 , for instance, it is specified that at the start of the execution of the startEngine() method there has to be fuel which can be burned.

```
context Car::startEngine():void
pre: fuelAmount > 0
```

Listing 5: Example of the *pre* construct in OCL.

The body (keyword *body*) of query operations can be specified in a single OCL expression. The evaluation of a query operation results in either a single value or a Set of values. In Listing 6 the query operation getAge() simply results in the age attribute of Car.

```
context Car::getAge():Integer
body: age
```

Listing 6: Example of the *body* construct in OCL.

OCL Expressions

A fundamental OCL expression refers to the keyword *self* which explicitly denotes the contextual instance of a type. For example in Listing 1, self indicates the contextual instance of a Car. However, the reference to self can be also omitted when referring to properties of a type like it is done in Listing 2.

OCL expressions are often built upon *navigations*. A navigation is performed when accessing attributes, operations or association ends. It is invoked by two operator types, the '.' operator and the '->' operator. The '.' navigation operator supports navigation from an object using attributes, operations or association ends. The '->' navigation operator supports navigation from a collection (see next section for further description) using attributes, operations, association ends or iterations. As aforementioned, in addition to the possibility to reference attributes and operations also association ends can be referenced. Thereby, one can navigate over an association on the model to refer to other model types and their properties starting from a specific model type. The referred model types themselves are also called navigations. Referencing the association end, can be done by calling the role name of the association. If the role name is missing, the name of the connected type can be used. Because navigations (in the sense of referencing association ends) are treated like attributes the dot notation is also used for accessing them. In order to avoid ambiguities, all navigation and attribute names must be unique. The value of a navigation is the Set of objects on the other side of the association end. If the multiplicity of the association end has a maximum of one, then the value of the expression is of the type the model type at the association end holds.

Figure 7 shows an example with two associations. A SportsCar has 4 wheels and each Wheel has an associated Info element. Listing 7 gives an example where a navigation expression for a contextual instance of SportsCar results in a Set. The navigation from SportsCar to its wheels is done via the role name. In contrast to this, the navigation shown in Listing 8 references the association end via the opposite model type name (i.e. Info). As it has a multiplicity of one, for a contextual instance of Wheel this expression results in the single element of type Info.

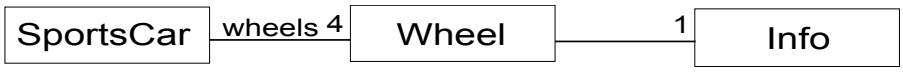

Figure 7: An association example

```
context SportsCar
self.wheels
```

Listing 7: Example of a *navigation* resulting in a Set.

```
context Wheel
self.Info
```

Listing 8: Example of an *navigation* resulting in a single value.

If more than one association with a multiplicity bigger than one is navigated, then per definition it results in a so called Bag (i.e. a Set which can have duplicate elements).

Another expression is the *if-then-else* expression which consists of a condition and two expressions. The result of the expression is either the evaluation of the expression in the *then* part or in the *else* part depending on the evaluation of the Boolean value in the condition (*if* part). The else part is mandatory because an OCL expression has to end in a value as it is a typed language. In Listing 9, the ageCategory is derived by an if-then-else expression which evaluates to "Young Car" if the age is smaller than 5 and else to "Old Car".

```
context Car::ageCategory:String
derive: if age < 5 then ''Young Car'' else ''Old Car'' endif
```

Listing 9: Example of an *if-then-else* expression in OCL.

A further expression is the *let-in* expression which is useful to define local variables which can be used in an OCL expression. Listing 10 has the same meaning like Listing 9 with the difference that it uses a let-in expression which defines the local variable young. However, variables declared in a let-in expression are only known in this expression.

```
context Car::ageCategory:String
derive: let young : Boolean = age < 5
                in young then ''Young Car'' else ''Old Car'' endif
```

Listing 10: Example of an *let-in* expression in OCL.

OCL Standard Library

In addition to the user defined model types such as the types SportsCar or Wheel which were previously introduced, OCL contains a number of predefined types. These types and their operations are specified in the *OCL Standard Library*. In the following, only an excerpt of the types and their corresponding operations are presented. A full overview of

Type	Values	Operations
Boolean	false, true	or, and, xor, not, =, <>, implies
Integer	-10, 0, 10, …	=, <>, <, >, <=, >=, +, -, *, /, mod(), div(), abs(), max(), min(), round(), floor()
Real	-1.5, 3.14, …	
String	„Carmen“	=, <>, concat(), size(), toLower(), toUpper(), substring()

Figure 8: Basic primitive types in OCL adapted from [10].

Operation	Description
count(o)	Number of occurrences of *o* in the collection (*self*)
execludes(o)	Is *o* not an element of the collection?
excludesAll(c)	Are all the elements of *c* not present in the collection?
includes(o)	Is *o* an element of the collection?
includesAll(c)	Are all the elements of *c* contained in the collection?
isEmpty()	Does the collection contain no element?
notEmpty()	Does the collection contain one or more elements?
size()	Number of elements in the collection
sum()	Addition of all elements in the collection

Figure 9: Standard Collection Operations adapted from [10].

them is given by the OCL Specification [31] or in a shorter version by the OCL Quick Reference [10].

The OCL Standard Library contains four *basic primitive types* which are the types *Boolean*, *Integer*, *Real* and *String* (see Figure 8). There exists an additional type called *UnlimitedNatural* which is used to encode non-Integer multiplicities like '*'. Each of these types provides different operations which can be invoked when referring to them. Examples are the 'and' operation for Boolean, the '+' operation for Integer and Real or the 'substring' operation for String.

Additionally, the OCL Standard Library specifies *collection types*. Thereby a collection is the abstract supertype of all collection types. Each collection type is parameterized by 'T' and can be concreted by replacing 'T' with Integer or SportsCar , for example, which creates a collection of Integer values respectively SportCars. Each occurrence of an object refers to an *element* of the collection. There are four collection types: *Set*, *OrderedSet*, *Bag* and *Sequence*. These types were already described in section 2.3.2. There are several *standard operations* which can be executed on Sets, OrderedSets, Bags and Sequences which are listed in Figure 9.

On the other hand, there are also operations which have different meaning respectively different preconditions depending on the collection type or are only invokable for particular collection types. Figure 10 lists such operations and also depicts for which types of collection they can be invoked. For instance, the *first* or the *last* operation can be only executed on OrderedSets and Sequences because they have an order, in contrast to Sets or Bags.

Operation	Set	OrderedSet	Bag	Sequence
=	O	O	O	O
<>	O	O	O	O
-	O	O		
append(o)		O		O
asBag()	O	O	O	O
asOrderedSet()	O	O	O	O
asSequence()	O	O	O	O
asSet()	O	O	O	O
at(i)*		O		O
excluding(o)	O	O	O	O
first()		O		O
flatten()	O	O	O	O
including(o)	O	O	O	O
indexOf(o)		O		O
insertAt(i, o)		O		O
intersection(c)	O		O	
last()		O		O
prepend(o)		O		O
subOrderedSet(l, u)		O		
subsequence(l, u)				O
symmetricDifference(c)	O			
union(c)	O	O	O	O

Figure 10: Collection Operations adapted from [10].

An example for a operation with a different meaning concerning the different collection types is the *equal* operation which can be called on all collection types. For a Set it evaluates to true if all elements in both Sets are the same. For a Bag elements must occur the same number of times. In case of a Sequence or a OrderedSet the order of the elements have to be the same. The *union* operation is an example for a collection operation that has different preconditions for different collection types. Bags and Sets are allowed to be unified. However, this operation is only callable for OrderedSets and Sequences if the argument is an OrderedSet respectively a Sequence too.

As opposed to collection operations which are called on the whole collection, *iteration operations* iterate over the elements of a collection and evaluate the expression for each of them. All iteration operations are presented in figure 11. Basically, all of these operations are based on the *iterate* operation. Iteration operations either result in a new collection (like *select*, *reject*,*collect*,*collectNested* and *sortedBy*), in a single element (like *any*) or in a Boolean value (like *forAll*, *exists*, *one*, *isUnique*).

The type on which all types (model types, primitive types and collection types) are derived from is called *OCLAny*. Hence, OCLAny is the supertype for all types of the OCL Standard Library. Because of this, operations defined for OCLAny are accessible for all types. Some of these operations are *oclIsTypeOf*, *oclIsKindOf*, *oclAsType* and *allInstances*.

The oclIsTypeOf operation evaluates to true if the type of the contextual instance is equal to the argument. The oclIsKindOf operation returns true if the type of the contextual instance is identical to the argument or one of the arguments's subtypes. Figure 3

Operation	Description
any(expr)	Returns any element for which *expr* is true
collect(expr)	Returns a collection that results from evaluating *expr* for each element of *self*
collectNested(expr)	Returns a collection of collections that result from evaluating *expr* for each element of *self*
exists(expr)	Has at least one element for which *expr* is true?
forAll(expr)	Is *expr* true for all elements?
isUnique(expr)	Does *expr* has unique value for all elements?
iterate(x: S; y: T| expr)	Iterates over all elements
one(expr)	Has only one element for which *expr* is true?
reject(expr)	Returns a collection containing all elements for which *expr* is false
select(expr)	Returns a collection containing all elements for which *expr* is true
sortedBy(expr)	Returns a collection containing all elements ordered by *expr*

Figure 11: Iterations Operations adapted from [10].

also contains the super/subtype relationship between Car and SportsCar. Assuming the contextual instance is an instance of the type SportsCar, "self.oclIsTypeOf(SportsCar)" would evaluate to true, but "self.oclIsTypeOf(Car)" would not. However, "self.oclIsKindOf(Car)" would return true.

By using the oclAsType operation a type cast can be performed. The argument of this operation can be of any model type defined in the model. The oclAsType operation changes the static type. Nevertheless, the object on which the operation is called results in the same object and its actual type remains the same. If the argument not conforms to the actual type at evaluation time (i.e. the oclIsKindOf operation evaluates to false), the operation results in *OCLInvalid*. Casting to subtypes provides visibility of features not defined in the context of an expression's static type. Casting to a supertype, however, does not allow the access to hidden or overridden features as the actual type stays the same.

Finally, the *allInstances* operation of the type OCLAny is a predefined operation for every model element which results in all the instances of the element and its subtypes.

2.4. Eclipse Plug-in Development

The tool which was extended by a multi-level aware OCL implementation as part of this thesis is called *Melanee* (see chapter 2.6). As this tool is an Eclipse plug-in, it is necessary to get the basic concepts of Eclipse plug-in development [12] which are explained in this chapter.

In general *Eclipse* "is a community for individuals and organizations who wish to collaborate on commercially-friendly open source software" [19]. The *Eclipse Foundation* is a non-profit corporation that hosts several projects. The Eclipse major project is the

Eclipse platform which is a multi-language software development environment serving also as Integrated Development Environment (IDE).

The Eclipse platform is constructed in a very modular design. It basically consists of a small kernel and several plug-ins which are building the modular units in Eclipse development. The kernel is responsible for managing the loading and execution of plug-ins. Each plug-in contributes to the functionality of the Eclipse platform. On one hand a plug-in may rely on services provided by other plug-ins, on the other hand it also can provide its services to other plug-ins. In order to allow this mutual extensibility, the concept of *extension points* and *extensions* was introduced. The idea behind this concept is that each plug-in can expose extension points which then can be extended by other plug-ins by providing an extension. The plug-in structure of the Eclipse platform enables developers to write their own plug-ins. Thereby, they can adapt , for example, the Eclipse IDE to their needs.

The minimal set of plug-ins necessary to create a so called rich client application is the *Eclipse Rich Client Platform* (RCP) [18]. This set is composed of two plug-ins, the *org.eclipse.ui* and *org.eclipse.core.runtime*, and their prerequisites. According to McAffer et al. [34], "rich clients support a high-quality end-user experience for a particular domain by providing rich native user interfaces (UIs) as well as high-speed local processing" [34]. They support functionalities like drag and drop, system clipboard, navigation as well as customization. The Eclipse RCP is a generic platform or framework for running applications which provides opportunities like a flexible UI paradigm, scalable UIs, extensible application, help support, error handling etc. [34]. The most prominent example for an application built on the Eclipse RCP is the Eclipse IDE which is written by the Eclipse team. However, also developers outside the Eclipse team can take advantage of the capabilities of the Eclipse RCP to develop their own rich client application (e.g. Melanee is based on the Eclipse RCP). They can achieve this by assembling a collection of plug-ins from the Eclipse base and elsewhere and adding the plug-ins they have implemented.

2.5. Eclipse Modeling Project

The *Eclipse Modeling Project* (EMP) is a project at Eclipse [21] which is a collection of frameworks, tools and standard technologies related to the MDD approach. EMP consists of five major components: *abstract syntax development*, *concrete syntax development*, *model transformations*, *model development tools* and *generative modeling technologies*. Figure 12 shows how these components are arranged in the overall EMP structure. At the core resides the *Eclipse Modeling Framework* (EMF) which provides abstract syntax development capabilities. Model transformation technologies are built on top of this core. These are divided into model-to-model (M2M) and model-to-text (M2T) technologies. The outer layer of EMP is formed by concrete syntax development technologies, like

the *Graphical Modeling Framework* (GMF) and *Textual Modeling Framework* (TMF). Model development tools and generative modeling technologies are surrounding the basic EMP structure. The model development tools focus on providing modeling tools based on industry standards like e.g. the MDA whereas the generative modeling technologies focus more on developing research-based components concerning MDD. In the following sections, EMF and GMF which were used to develop Melanee and the *Eclipse OCL Project* which was utilized for implementing the deep OCL dialect are described in more detail.

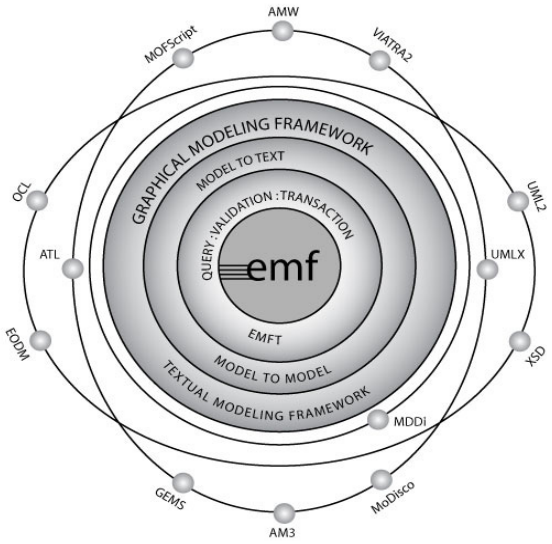

Figure 12: The structure of EMP [25].

2.5.1. Eclipse Modeling Framework

The *Eclipse Modeling Framework* (EMF) [20] is a modeling framework supporting MDD activities which "exploits the facilities of Eclipse" [38]. With respect to Gronback [25], the EMF serves for developing the abstract syntax of DSLs. The meta meta-model which is used to create such meta-models in EMF is called *Ecore* which is in accordance with the EMOF. As EMOF is a subset of the MOF, Ecore can be perceived as MOF compliant. Based on Ecore, EMF enables the definition as well as the manipulation of meta-models by using a generic meta-model editor, the serialization of meta-models, the generation of Java implementation classes created from EMF models and much more. The importance of EMF in the MDD environment is highlighted by Stahl et al. [37] which perceive EMF as a de facto industry standard when building MDD tools.

2.5.2. Graphical Modeling Framework

The *Graphical Modeling Framework* (GMF) [17] originated from the need for an easier way to create graphical editors using the *Graphical Editing Framework* (GEF) which are

based on EMF models [25]. The GEF itself provides the technology to create graphical editors and views within Eclipse. With the help of GMF such graphical editors can be created automatically for EMF meta-models. The process of this creation is roughly described by Stahl et al. [37]. At first, a meta-model is defined with the help of EMF. After this, an additional model is created describing how the graphical editor should look like and behave. Then by using these two models, the GMF generator is able to create the graphical editor. Finally, some custom visualization or specific behavior can be added to the generated editor.

2.5.3. Eclipse OCL Project

The Eclipse OCL Project [16] is part of the MDT project and is an OCL implementation of the OMG's OCL standard for models based on the EMF. This project consists of a core component and additional OCL examples and editor components.

The core component helps integrating OCL in EMF based models and has several capabilities. Besides a generic abstract syntax model API (independent of any specific meta-model) it also makes available a particular Ecore as well as an UML implementation of the OCL abstract syntax (see section 2.3.2). Based on this, it specifies interfaces for the definition, parsing and evaluation of OCL queries and constraints on Ecore and UML models, but also any other model. Furthermore, the Eclipse OCL Project allows developers to extend the API for parsing and evaluating OCL queries and constraints, so that they can implement a custom behavior when parsing and evaluating OCL expressions. Moreover, a visitor API is provided for inspecting and analyzing the abstract syntax tree of parsed OCL expressions. Additionally, Eclipse OCL supports the serialization of parsed OCL expressions so that they can be persisted.

The additional provided examples and components enable an interactive OCL support when building EMF models. For example, an Xtext editor for specifying OCL expressions and an interactive OCL console for the evaluation of OCL expressions on Ecore or UML models is provided.

2.6. Melanee

The *Multi-level Modeling and Ontology Engineering Environment* (Melanee) ([35], [5], [32], [23]) is a LML (see section 2.2.2) editor enabling the creation and manipulation of LML diagrams like the one in Figure 3. In essence, Melanee comprises real-time interactive deep modeling. The tool was developed by the Chair of Software Engineering of the University of Mannheim.

The PLM which serves as meta-model or abstract syntax was realized in EMF (see section 2.5.1). It is a Ecore model and forms the meta-model for the LML. Furthermore,

meta-model operations were implemented as OCL operations. Those and also meta-model attributes are accessible in the interactive OCL console which allows live queries on the LML model. The visualization respectively the concrete syntax of the editor was implemented in GMF (see section 2.5.2).

Melanee consists of several software components which add additional capabilities in addition to the pure creation of LML models. *Reasoning and checking services* help to validate ontologies for correctness and also to infer types from instances or inheritance relationships between elements of the model. The LML can be defined graphically (*graphical DSL*) or textually (*textual DSL*). Moreover, these can also be used simultaneously. Regarding the graphical DSL capability, the rendering of elements can be customized to create a domain-specific notation or the general-purpose LML rendering can be applied. This approach is referred as *symbiotic language support*. The *emendation* mechanism helps to automatically keep the ontology consistent when a model element is changed by observing the effects to the whole ontology after a single element is manipulated. Finally, Melanee also provides a *multi-level aware transformation support* so that multi-level to multi-level as well as multi-level to two-level transformations can be achieved.

This tool is implemented on top of the Eclipse framework so it builds basically a set of Eclipse plug-ins and is based on the Eclispe RCP (see section 2.4). Because of this, Melanee is extensible through third party technologies and can be adapted and customized individually.

3. Interpretation and Application of OCL in Deep Models

OCL is a powerful language for the query of model elements and their properties and for the definition of constraints on model elements. An introduction of OCL in general is given in section 2.3. The OCL specification [31] is tailored for the application of OCL in two-level models.

However, as the goal of this thesis is to develop a level-agnostic OCL console, the interpretation and the application of OCL in deep models have to be examined in order to elaborate a *deep OCL dialect*. This is done in this chapter. An example of a deep LML model depicted in Figure 13 helps to describe the principles and concepts of a deep OCL dialect. In essence, the ontology of the example describes the relation of cars between its wheels and between the company which produces them. The company itself employs employees. Due to the lack of space, the obligatory wheels for the clabjects MyPorscheBoxster and MyPorscheCayenne in 02 are omitted here. The example is further introduced in the discourse of this chapter when describing the particular artifacts of a deep OCL dialect.

3.1. Requirements for the Evaluation of OCL Expressions in a Deep OCL Dialect

OCL expressions form the basis of OCL as queries as well as constraints are built upon them. In order to create a profound and consistent specification of a deep OCL dialect, the peculiarities of deep modeling environments have to be considered.

These peculiarities evoke requirements for the evaluation of OCL expressions in a deep OCL dialect which are examined in this section.

3.1.1. Dual Facet of Clabjects

When evaluating OCL expressions, it is very important to consider the dual facets of model elements in a deep modeling environment. In the LML, this duality of model elements is represented by a unified concept, called clabject. A clabject captures both, the type and instance facet of a model element.

However, in the two-level modeling case a model element is either a type or an instance but never both. In conformance to this, OCL proposes the terms of *contextual type* and *contextual instance* (see section 2.3.3). OCL expressions are defined for a conceptual type and evaluated on a particular instance of this type. This works fine in a two-level model as properties are defined on the type level and only enriched with specific values on the instance level which then can be queried and evaluated.

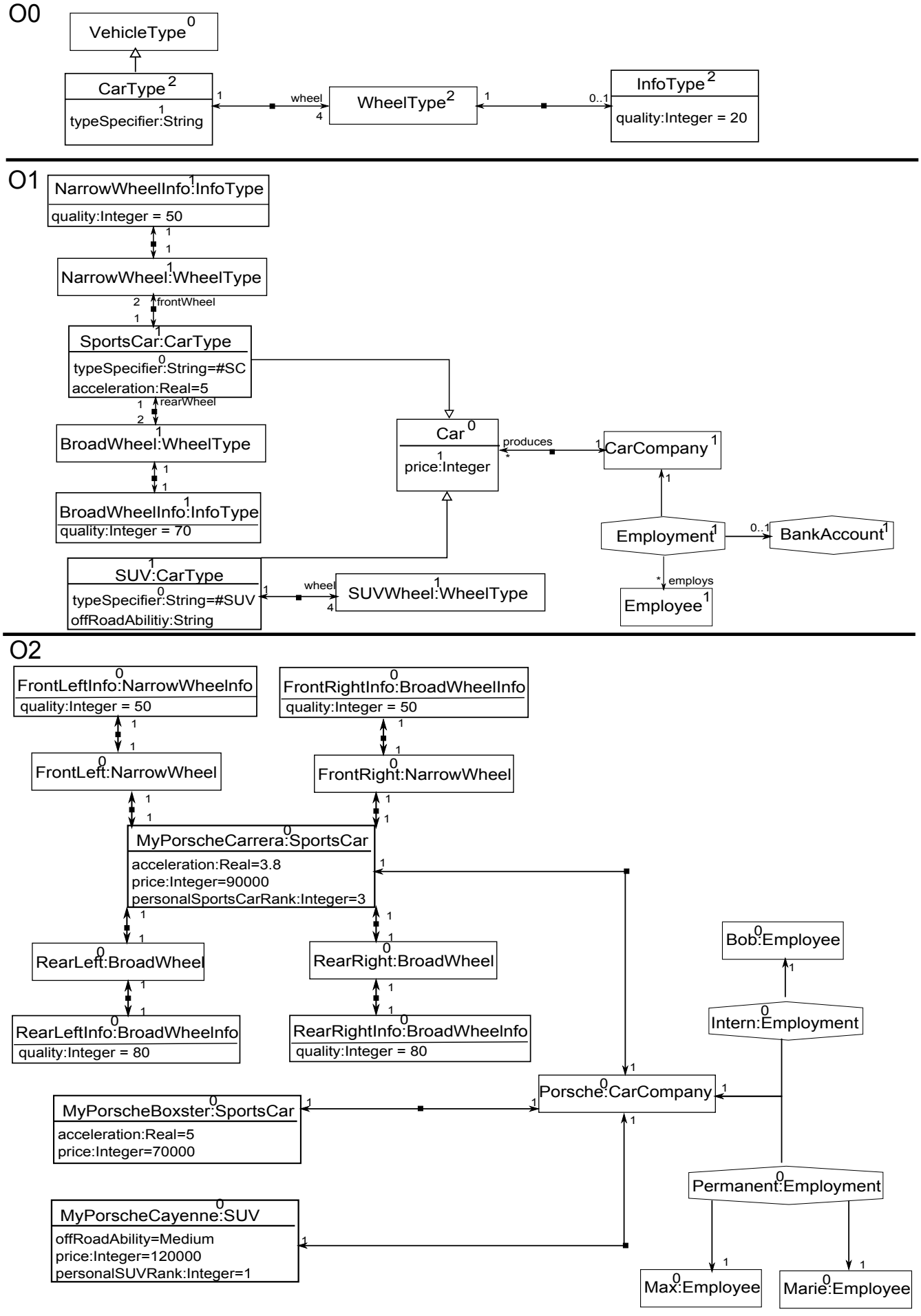

Figure 13: Example of a deep model modeled with the LML.

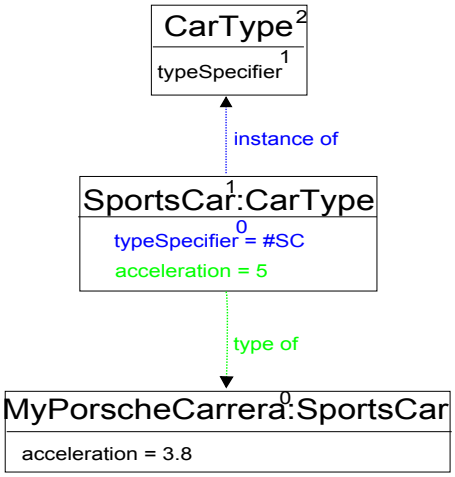

Figure 14: Dual facet of model elements.

But as mentioned before, the strict separation regarding a model element as either a type or an instance is not possible in deep models. This is illustrated in Figure 14. With respect to the entity SportsCar, on the one hand it is a type of MyPorscheCarrera but on the other hand also an instance of a CarType. From an instance point of view, the typeSpecifier attribute specified in CarType is instantiated by attaching a value to it ('#SC'). From a type's viewpoint, the attribute acceleration is defined and additionally a value of 5 is already specified here.

When now the context is set to SportCar, this would mean that the contextual type is also SportCar which is in accordance with Warmer and Kleppe [39] who claim that "when the context itself is a type, the context is equal to the contextual type". The SportsCar also holds an instance facet, however, as it can be regarded as an instance of CarType. Thus, it has to be discussed if the notion of contextual type is still valid here.

Besides this difficulty, let us assume that the OCL expression "self.acceleration" should be evaluated on SportCar. Referring to Warmer and Kleppe [39] as well as to the OCL Specification [31], this expression would be evaluated on single objects respectively instances of the contextual type. The contextual type in this case would be SportsCar and the evaluation of the expression would be conducted on its contextual instances, e.g. on MyPorscheCarrera. In this regard, the question arises how to get the value of the acceleration attribute of SportsCar because this value is already attached to the attribute when it is just defined in SportsCar and hence it is not accessible on its instances.

The presented example use cases demonstrate that the OCL concepts of contextual type and conceptual instance are difficult to apply in a deep modeling environment. Therefore, a revised concept concerning the evaluation context of OCL expressions which is adapted to the dual facets of clabject has to be elaborated in a deep OCL dialect.

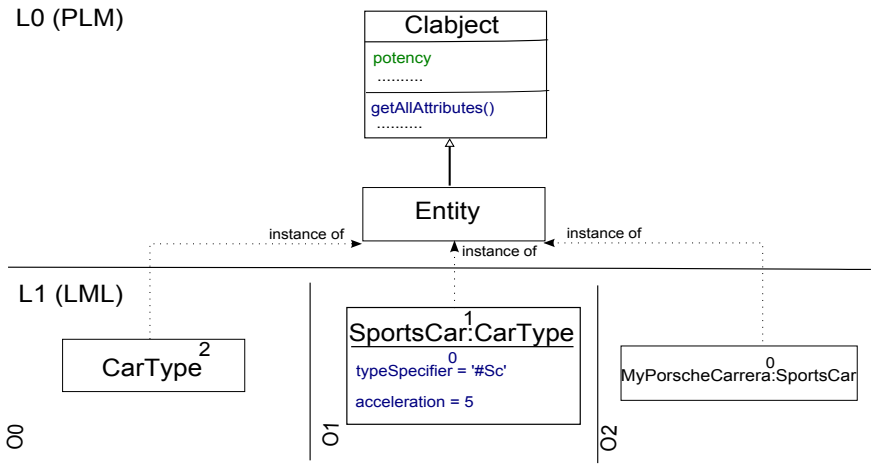

Figure 15: Linguistic dimension example.

3.1.2. Ontological and Linguistic Dimension

The previous section described the dual facet of model elements in the ontological dimension which captures the actual problem domain. In the deep modeling approach, however, there exists another dimension called the linguistic dimension.

Linguistic types describe the structure, i.e. the abstract syntax, of model elements independent from their ontological level. This is visualized in Figure 15. CarType, SportsCar and MyPorscheCarrera are all linguistic instances of the *entity meta-type* which itself is a subtype of the *clabject meta-type*. These and other meta-types are defined in the PLM which resides in the linguistic layer L0. A clabject has attributes such as potency and also methods such as getAllAttributes which, as the name is indicating, delivers all attributes of a particular clabject instance. In contrast to the ontological dimension (see previous chapter), the linguistic instance-of relationship between L0 and L1 is of a two-level nature and so the OCL principle of contextual type and contextual instance is applicable here.

Assuming the context is set to SportsCar, from a linguistic perspective the contextual type would be the entity type. SportsCar has a concrete value for potency (indicated by green color) and also a set of concrete attributes (indicated by blue color). Thus, from the linguistic perspective, SportsCar can be regarded as contextual instance of entity so that OCL expressions would be evaluated here.

It is of particular importance to provide the possibility to query not only ontological properties but also linguistic ones in a deep OCL dialect. Therefore, a solution has to be found which allows to switch from ontological to linguistic perspective.

3.1.3. Deep Characterization

In the ontological dimension of the OCA architecture an arbitrary number of levels is allowed which helps to reflect a specific problem domain.

In chapter 2.2 the concept of Deep Characterization is introduced. This concept refers to the existence of multiple levels in an ontology and permits elements not to control only elements the level below but also elements further levels below. Hence, model elements can also have *deep instances* in addition to instances . Considering the example in Figure 14, a CarType has a potency of two and thus can be instantiated over two more levels. SportsCar is placed a level below and is an instance of CarType. MyPorscheCarrera is placed in the O2. It is an instance of SportsCar but also can be perceived as deep instance of CarType. Taking the opposite viewpoint, SportsCar can be considered as type of MyPorscheCarrera but CarType as deep type of MyPorscheCarrera.

With respect to a deep OCL dialect, the concept of Deep Characterization has to be taken into account regarding the operations *oclIsTypeOf*, *oclIsKindOf*, *oclAsType* and *allInstances* of the *OCLAny* type (see section 2.3.3). Those operations are suitable and applicable in the context of two-level modeling as there is only one type and one instance level. In deep modeling environments, however, Deep Characterization and thus the existence of deep types and instances in the ontological dimension has to be taken into consideration when specifying these operations for a deep OCL dialect.

3.1.4. Navigation in the LML

An important aspect of OCL is navigation on associations between model elements. Starting from a specific element, navigation enables to refer to other elements and the corresponding properties.

In the LML, associations are represented by the concept of *connections*. They are defined in the PLM as subtype of a clabject. Therefore, they also can have attributes and methods. Connections can refer to an arbitrary number of particpants (i.e. clabjects) which are connected by a related particpation holding role name as well as lower and upper multiplicity. In comparison to the UML, connections can be considered as association classes.

However, the standard OCL specification is not aware of the connection construct. Hence, a deep OCL dialect has to take connections into account. First of all, a specification has to be developed of how navigation can be conducted syntactically and semantically. Figure 16 presents the entities CarCompany and Employee which are connected by the connection Employment, visualized in exploded form. In this example, three different kinds of navigations are visualized.

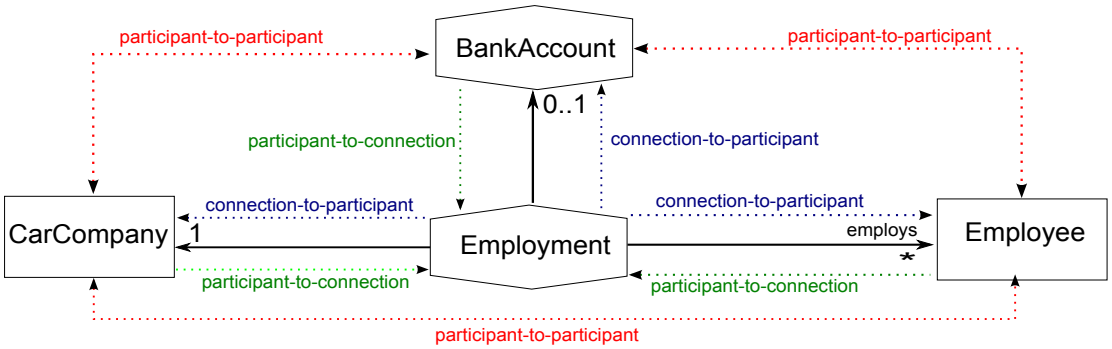

Figure 16: Three different kinds of navigation.

The navigations between CarCompany and Employee, CarCompany and BankAccount as well as BankAccount and Employee represent examples of a *particpant-to-particpant* navigation. Examples for *connection-to-particpant* navigations are the ones from Employment to CarCompany, from Employment to BankAccount as well as from Employment to Employee. The last kind of navigation is the *particpant-to-connection* navigation. Examples for this kind of navigation are the navigations from CarCompany to Employment, from BankAccount to Employment as well as from Employee to Employment. In regard to a deep OCL dialect, it has to be elaborated how these kinds of navigation can be performed.

Based on this, a deep OCL dialect also has to specify in which cases a navigation evaluates to which results. This means that this dialect has to define under which conditions a navigation results in a single clabject or a collection of clabjects and in the latter case also what kind of collection it is.

Furthermore, the dual facet of clabjects has to be considered. As mentioned before, connections are clabjects and thus also have a type and instance facet. Figure 17 visualizes this fact with an example.

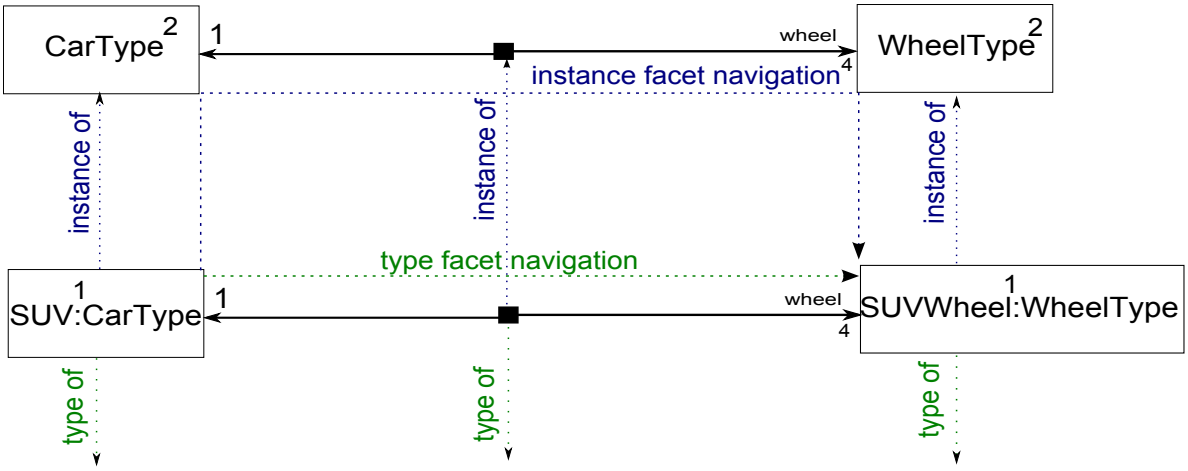

Figure 17: Navigation on the type and instance facet.

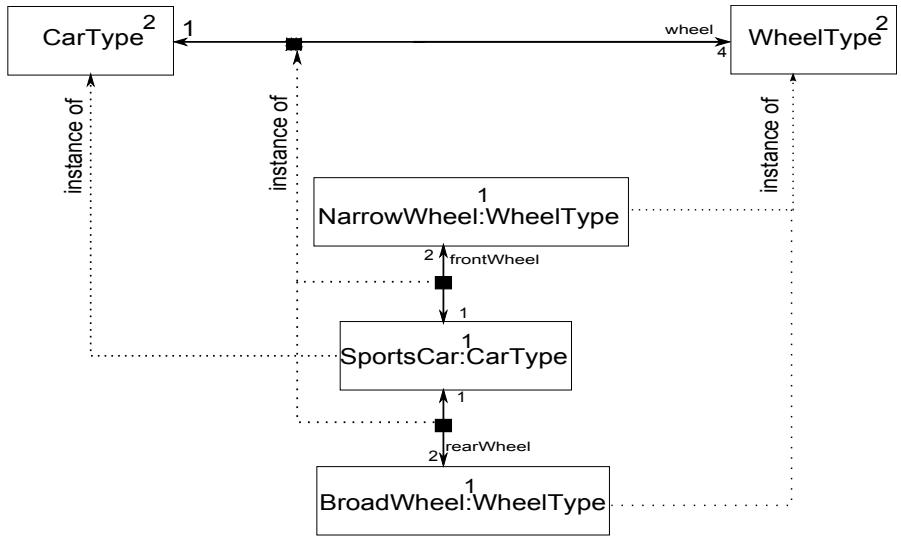

Figure 18: Example of 2 instantiations of a connection with different role names.

In the example two connections are shown (both in dotted form). The first connection associates a SUV with SUVWheel. SUV, SUVWheel and its connection all have a type facet. All of them are also instances, however, as SUV is an instance of CarType, SUVWheel is an instance of WheelType and the connection is an instance of the shown connection the level above. This second connection associates a CarType with WheelType.

When now navigating from SUV to SUVWheel, this navigation could be perceived as a *type facet navigation* or an *instance facet navigation* in a deep OCL dialect. Therefore, it has to be elaborated how these two different facet viewpoints concerning navigation can be conducted syntactically and how these affect the evaluation of the navigation.

Furthermore, a deep OCL dialect has to be aware of the fact that connections can have multiple instances on the level below and that their names as well as the role names of their participations can be changed when they are instantiated. An example for this case is illustrated in Figure 18.

Here, a connection is instantiated twice. First a connection is instantiated which represents the connection between SportsCar and NarrowWheel which are in the front (role name frontWheel). Moreover, a connection is instantiated which represents the connection between SportsCar and BroadWheel which are in the rear (role name rearWheel). The question arises if it is possible to not only navigate from the SportsCar to the front respectively rear wheels but to all its wheels. This question relates to the aforementioned type facet and instance facet navigation and has also to be answered in the specification of a deep OCL dialect.

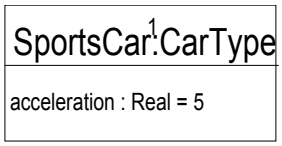

Figure 19: SportsCar entity with attribute acceleration of type Real.

3.1.5. Datatypes in the LML

According to the OCL Standard Library (see section 2.3.3), there are four primitive basic datatypes known in the OCL which are *Boolean*, *Integer*, *Real* and *String*.

These types are also available when setting types of attributes for clabjects in a LML model. In the current PLM specification, however, values of attributes are defined as String independent from their actual ontological datatype.

Figure 19, for instance, shows the SportsCar entity which holds an acceleration attribute of the type Real. Assuming the OCL expression "self.acceleration" should be evaluated on a SportsCar, an ambiguity arises. Regarding the PLM specification, the attribute value is defined as String (linguistic definition). Nevertheless, the actual ontological datatype of acceleration in this example is Real (ontological definition). A deep OCL dialect has to take this into consideration and has to specify to which datatype OCL expressions like the one above evaluate.

3.2. Proposals for the Evaluation of OCL Expressions in a Deep OCL Dialect

In this chapter a specification for a deep OCL dialect concerning the evaluation of OCL expressions is elaborated. The specification adheres to the requirements for the evaluation of OCL expressions which were described in the previous chapter and also tries to follow as much of the principles and definitions of the OCL specification [31] as possible.

3.2.1. Supporting Dual Facet of Clabjects

Section 3.1.1 discussed the difficulty of applying the OCL concepts of contextual type and contextual instance in a deep modeling environment. Therefore, a deep OCL dialect unifies those two concepts by referring to a *conceptual clabject* when evaluating OCL expressions.

As a consequence, when the context is set to a specific clabject all attributes and methods of the clabject can be queried by OCL expressions regardless of whether those were instantiated or were just defined in the clabject.

In regard to Figure 14 for example, the context could be set to SportsCar and the value of typeSpecifier which is an instantiated attribute (defined already in CarType) and also the value of acceleration which was just specified in SportsCar can be queried.

3.2.2. Accessing Ontological and Linguistic Dimension

As described in section 2.2.2, clabjects in the LML reside in the OCA and therefore have two dimensions, an ontological and a linguistic one. A deep OCL dialect should be able to adopt the perspective of both dimensions which allows the query of ontological but also linguistic properties with OCL expressions.

The default perspective of a deep OCL dialect should be the ontological one, as the focus of deep modeling lies on representing an ontological problem domain. A switch from the ontological to the linguistic perspective in a deep OCL dialect can be done by invoking the "_l_" property of a clabject.

Assuming that the context is set to SportsCar, per default all ontological properties of SportsCar are accessible (for example typeSpecifier). However, the properties of the linguistic type (i.e. entity) can not be called. In order to obtain them, a perspective switch has to be performed. This is done, for instance, in Listing 11.

```
context SportsCar
self._l_.potency -- 1
```

Listing 11: Example of accessing the linguistic dimension in a deep OCL dialect.

In this case, the potency, a linguistic attribute defined in the clabject meta-type (and therefore also available in the entity meta-type), of SportsCar is queried which is 1.

Hence, the proposed syntax of the deep OCL dialect provides a flexible mechanism of accessing ontological and linguistic dimension which allows the query of all properties, be it ontological or linguistic ones, of clabjects in the OCA.

3.2.3. Application of OCLAny Operations

In this section, the effects of deep modeling environments on the interpretation and application on the operations *oclIsTypeOf*, *oclIsKindOf*, *oclAsType* and *allInstances* of the type *OCLAny* are discussed. Hereby, the existence of multiple levels in the ontological dimension as well as the existence of the two dimensions in the OCA has to be taken into account. In the following, each of these operations and their behavior in a deep OCL dialect with regard to the aforementioned peculiarities are described.

oclIsTypeOf/oclIsKindOf

Concerning the ontological dimension, the oclIsTypeOf and the oclIsKindOf operation can be applied in a deep OCL dialect in a two-level window and thus are interpreted like in the two-level modeling case which was described in section 2.3.3. In Figure 20, for

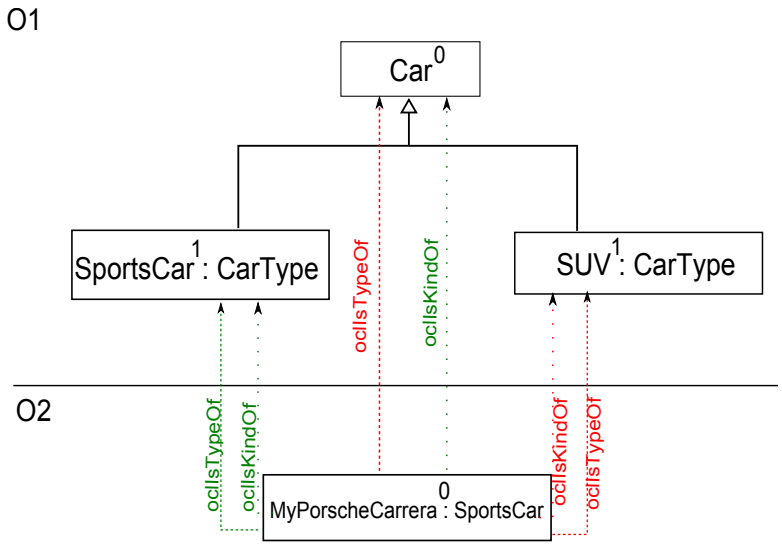

Figure 20: Example usage of oclIsTypeOf/oclIsKindOf in a deep OCL dialect.

example, two ontological levels (01 and 02) are shown. In O1, there are three clabjects which are SportsCar and SUV as well as their supertype Car. In 02 resides the clabject MyPorscheCarrera, an instance of SportsCar.

Referring to MyPorscheCarrera as context, the operations oclIsTypeOf and oclIsKindOf are invoked with each of the other clabjects in this example as argument. A call of the oclIsTypeOf operation only evaluates to true with the argument SportsCar (indicated by the green color). In the two other cases the result is false (indicated by the red color). Invoking the oclIsKindOf operation, evaluates to true when passing SportsCar and Car as argument, but to false when passing SUV.

These two operations, however, can only be used in a two-level window in a deep model. As described in section 3.1.3, besides ontological types defined in the direct level above, clabjects can also have deep ontological types defined the levels further above. Thus, it should be also possible to check deep types in a deep OCL dialect. Because of this, the operations *oclIsDeepTypeOf* and *oclIsDeepKindOf* are introduced in the dialect.

In Figure 21, an example usage of these operations is presented, where three ontological levels are visualized. In the O0 are the clabjects CarType and VehicleType whereas CarType is a subtype of VehicleType. The entity MyPorscheCarrera in 02 is a deep instance of CarType which can be instantiated in O1 and O2.

Referring now to MyPorscheCarrera as context, the oclIsDeepTypeOf and oclIsDeepKindOf operation can be invoked to check its deep types. In this example, the oclIsDeepType operation is called with the argument CarType and VehicleType. In the first case it evaluates to true, in the second case to false. The same is done for the oclIsDeepKind operation. For both arguments it evaluates to true.

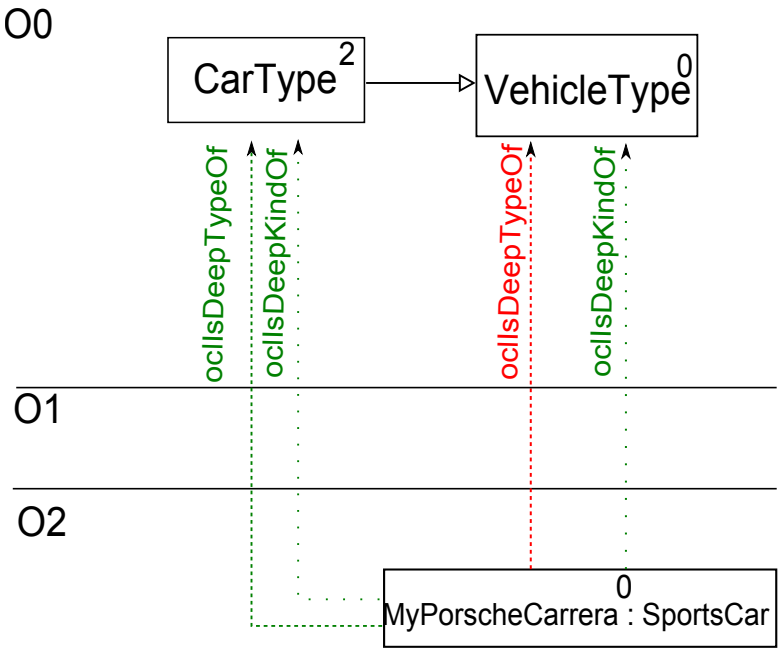

Figure 21: Example usage of oclIsDeepTypeOf/oclIsDeepKindOf in a deep OCL dialect.

Moreover, the oclIsTypeOf and oclIsKindOf operation can also be applied regarding the linguistic dimension. In the deep OCL dialect, this is done by first accessing the linguistic dimension with "_l_" (see previous chapter) and then invoking one of the two operations. An example is shown in Figure 22.

In contrast to the previous examples, the linguistic dimension is displayed with the two linguistic levels L0 and L1. In the L0 are shown the linguistic types Entity, Connection and its supertype Clabject. In the L1 the MyPorscheCarrera entity is presented. Ontological levels are not shown in this example.

Assuming the context is SportsCar, the oclIsTypeOf and oclIsKindOf is invoked with each of the presented linguistic types as argument. The linguistic perspective is adopted by invoking "_l_" before. The oclIsTypeOf operation evaluates to true only with the argument Entity. The oclIsKindOf operation, however, evaluates also to true with the argument Clabject.

oclAsType

The oclAsType operation is, like the operations oclIsTypeOf and oclIsKindOf, also only applicable in a two-level window of a deep model representing an ontological domain. The operation has the same meaning like defined in the OCL specification [31]. This means that the static type can be changed by this operation while the actual time remains the same. If the argument not conforms to the actual type, i.e. the oclIsKindOf operation evaluates to false, OCLInvalid is returned.

With respect to Figure 20, for instance, in all cases in which oclIsKindOf evaluate to true the oclAsType operation and so a type cast can be accomplished successfully. This

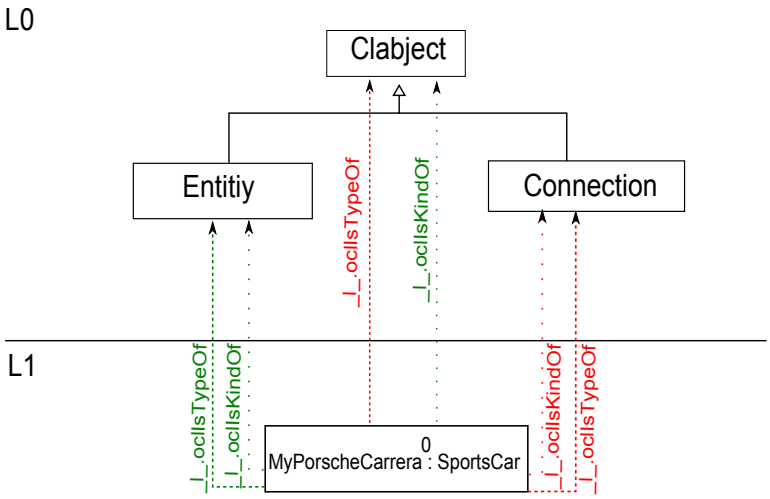

Figure 22: Example usage of oclIsTypeOf/oclIsKindOf in the linguistic dimension in a deep OCL dialect.

is illustrated by Listing 12. In this case, the cast to Car shows a supertype cast. A cast to a subtype cannot be shown yet as the concept of navigation in a deep OCL dialect needs to be introduced first. Therefore, an example of this case is given in section 3.2.4.

```
context MyPorscheCarrera
self.oclAsType(SportsCar) -- valid
self.oclAsType(Car) -- valid
self.oclAsType(SUV) -- OCLInvalid
```

Listing 12: Example of using the operation oclAsType in a deep OCL dialect.

In order to cast to a deep ontological type, an additional operation needs to be introduced for the deep OCL dialect called *oclAsDeepType*. In accordance to the two-level case, casting to a deep type only causes a change of the static type while the actual type stays the same. If the argument not conforms to the actual type, i.e. oclIsDeepKindOf evaluates to false, OCLInvalid is returned.

Regarding Figure 21, a deep type cast with oclAsDeepType can be conducted in both cases as SportsCar is a deep kind of both, CarType and VehicleType. Listing 13 shows how the operation is invoked. It also shows that a deep type cast to SportsCar evaluates to OCLInvalid. This is because MyPorscheCarrera is not a deep kind of Sportscar, albeit it is a kind of SportsCar.

```
context SportsCar
self.oclAsDeepType(CarType) -- valid
self.oclAsDeepType(VehicleType) -- valid
self.oclAsDeepType(Sportscar) -- OCLInvalid
```

Listing 13: Example of using the operation oclAsDeepType in a deep OCL dialect.

In compliance to the operations oclIsTypeOf and oclIsKindOf, the oclAsTypeOf operation can also be used in the linguistic dimension. This is accomplished by first obtaining the linguistic perspective by "_l_" and then invoking the oclAsType operation.

Respecting Figure 22, this means that in all cases in which "_l_.oclIsKindOf" evaluates to true a type cast can be done, otherwise OCLInvalid is returned. Listing 14 shows the particular calls of "_l_.oclAsType" and its associated results.

```
context MyPorscheCarrera
self._l_.oclAsType(Entity) -- valid
self._l_.oclAsType(Clabject) -- valid
self._l_.oclAsType(Connection) -- OCLInvalid
```

Listing 14: Example of using the operation oclAsType in the linguistic dimension in a deep OCL dialect.

allInstances

The meaning of the allInstances operation needs to be discussed in a deep modeling environment. By definition, the allInstances operation can be invoked for every element in a model and it returns a set of all instances of the element and all its subtypes (see section 2.3.3).

Applying this to a deep OCL dialect with regard to the ontological dimension means that the allInstances operation returns a set of all instances of a clabject and all its subtypes the direct level below. Thus, when invoking, for example, allInstances on CarType, this returns SportsCar and SUV which is shown in Listing 15.

```
CarType::allInstances() -- SportsCar and SUV
```

Listing 15: Example of using the operation allInstances referring to the ontological dimension in a deep OCL dialect.

However, the query of only ontological instances of a clabject might be not enough in a deep model as every clabject also can have deep ontological instances, which was indicated in section 3.1.3. Therefore, another operation is introduced for each clabject concerning the ontological dimension in a deep OCL dialect which is called *allDeepInstances*. This operation returns deep instances of a clabject, i.e. the instances of the clabject which are instantiated on a level which is more than one below the level where the clabject itself was instantiated. Listing 16, for instance, shows that invoking allDeepInstances on CarType returns MyPorscheCarrera, MyPorscheBoxster and MyPorscheCayenne which are in the 02 (see Figure 13).

```
CarType::allDeepInstances() -- Set{MyPorscheCarrera, MyPorscheBoxster,
    MyPorschCayenne}
CarType::allDeepInstances(1) -- Set{MyPorscheCarrera, MyPorscheBoxster,
    MyPorschCayenne }
```

Listing 16: Example of using the operation allDeepInstances referring to the ontological dimension in a deep OCL dialect.

When there are more than three levels in a deep model, a clabject can also have deep instances over more than one level. Hence, deep instances have a certain *degree*. So, for example, MyPorscheCarrera, MyPorscheBoxster and MyPorscheCayenne are deep instances of CarType of degree one. If there would be another ontological level 03 below 02 with deep instances of CarType, these would have a degree of two. This degree can be passed as argument to the allDeepInstances operation like it is done, for instance, in the second expression of Listing 16 which have the same meaning like the first expression in this case. It is also possible to pass two arguments, whereas the first argument specifies the lower degree and the second argument specifies the upper degree so that deep instances of the clabject are returned whose degree is within the specified degree range.

Furthermore, the allInstances but not the allDeepInstances operation can be invoked also on linguistic meta-types. Calling "Clabject::allInstances()", for example, would return all clabjects of a whole ontology regardless their ontological level.

3.2.4. Conducting Navigation in Deep Models with OCL

Section 3.1.4 reveals several issues concerning navigation in a deep model a deep OCL dialect must be aware of. This section aims to solve these issues by specifying how navigation can be interpreted and applied in a deep OCL dialect.

Navigation Kinds

Figure 16 visualizes the different navigation kinds in the LML which are participant-to-participant, connection-to-participant and participant-to-connection.

Navigation from a participant of a connection to another participant can be done either by referring to the role name of the participation or to the name of the associated participant. Listing 17 shows both variants when navigating from a CarCompany to an Employee. But in case the role name is missing, the name of the participant has to be used in order to navigate.

```
context CarCompany
self.employs -- Employee
self.Employee -- Employee
```

Listing 17: Example for a participant-to-participant navigation.

A connection-to-participant navigation is done like in the participant-to-participant case either by the role name or the name of the participant. Listing 18 presents these variants by navigating from Employment to Employee.

```
context Employment
self.employs -- Employee
self.Employee -- Employee
```

Listing 18: Example for a connection-to-participant navigation.

A navigation from a participant to its associated connection can be conducted by referring to the name of the connection. Listing 19 presents an example where a navigation from the entity CarCompany to the connection Employment is performed.

```
context CarCompany
self.Employment -- Employment
```

Listing 19: Example for a participant-to-connection navigation.

Result Types

With respect to the OCL specification [31], navigation can either result in a single model element or a Set of model elements. In accordance to this, the navigation in a deep OCL dialect either results in a single clabject or in a Set of clabjects.

The result type of a participant-to-participant and connection-to-participant connection depends on the multiplicity of the associated participation. If its multiplicity is at most one, the result conforms to the clabject to which the navigation refers. If the multiplicity is greater than one, the result type is a Set. Hence, with regard to Figure 16, for example, a navigation from CarCompany to Employee results in a Employee Set whereas the navigation from Employment to CarCompany results in a single CarCompany.

When navigating from a participation to a connection, the multiplicities of the other participations of the navigated connection have to be taken into account. If any of the other participations has a multiplicity higher than one, the result type is a Set, otherwise it is a single clabject. A navigation such as the one from CarCompany to Employment leads to a Set of Employments as the participation from Employment to Employee has a '*' multiplicity. As opposed to this, a navigation from Employee to Employment results in a single Employment as the participation from Employment to BankAccount as well as the one from Employment to CarCompany has a multiplicity of at most one.

When a navigation is performed over more than one clabject with a multiplicity greater than one, the result is a Bag.

Type and Instance Facet Navigation

Figure 17 reflects the duality of type and instance facet when navigating in a deep OCL dialect with an example navigation from SUV to SUVWheel. In the following, it is described how type and instance facet navigation can be realized in a deep OCL dialect.

A type facet navigation is done by simply conducting a navigation like for instance the one from SUV to SUVWheel shown in Listing 20.

```
context SUV
self.wheels -- Set{SUVWheel}
self.SUVWheel -- Set{SUVWheel}
```

Listing 20: Example for a type facet navigation.

As mentioned before, the result evaluates to a Set of SUVWheels because the multiplicity is four and thus higher than one. Nevertheless, the result is a Set containing only one SUVWheel as navigation is done from a type facet point of view in this case and there is actually only one SUVWheel entity modeled on this level, serving as type for entities the levels below.

In addition to this type facet, however, the connection between SUV and SUVWheel forms also an instantiation of the connection between CarType and WheelType the level above. Taking an instance point of view, a SUV with four SUVWheels is instantiated here. Therefore, in a deep OCL dialect also an instance facet navigation has to be allowed. In order to do this, a so called *level cast* is introduced in the deep OCL dialect. A level cast enables a clabject to access navigation properties of its types and deep types which makes instance facet navigation possible. But it is important to consider that a level cast never changes the actual contextual clabject. Listing 21 shows an example where an instance facet navigation from SUV to SUVWheel type is done.

```
context SUV
self._CarType_.wheels -- Bag{SUVWheel,SUVWheel,SUVWheel,SUVWheel}
self._CarType_.WheelType -- Bag{SUVWheel,SUVWheel,SUVWheel,SUVWheel}
```

Listing 21: Example for an instance facet navigation.

In this case, a level cast to CarType with the help of the notion "_CarType_" is applied. In general, the syntax of a level cast consists of a clabject name surrounded by '_' (see defintion 1). The instance facet navigation in this example evaluates to a Bag of four SUVWheel entities.

Definition 1 *<level cast> ::= '_' <ClabjectName> '_'*

In respect of section 3.1.4 and the example in Figure 18, the question arose how navigation from a SportsCar to all its wheels can be done. In this case, two instantiations of the connection between CarType and WheelType exist which also have different role names (frontWheels and rearWheels). Now, this can be answered with the help of the level cast construct and an accompanying instance facet navigation. In Listing 22, for example, the first instance facet navigation from SportsCar results in a Bag of two NarrowWheels and two BroadWheels. In the second expression only the NarrowWheels of this Bag are selected and thus the result is a Bag of two NarrowWheels.

```
context SportsCar
self._CarType_.wheels -- Bag{NarrowWheel, NarrowWheel, BroadWheel,
    BroadWheel}
self._CarType_.wheels->select(a|a=NarrowWheel) --  Bag{NarrowWheel,
    NarrowWheel}
```

Listing 22: Examples for a instance facet navigation having two instantiations of a connection.

It has to be regarded that the result in the expressions in Listing 21 and Listing 22 are Bags, not Sets. Although the simple navigation from CarType to its wheels would result in a Set, in these cases it results in a Bag. This is because an instance facet navigation is used here which could return identical clabjects such as 2 NarrowWheels in Listing 18. This requires a Bag as this collection type can store identical elements. Thus, the result of an instance facet navigation in a deep OCL dialect is always a Bag if the multiplicity of the opposite participation is bigger than one.

In Listing 23 the context is set to an instance of SportsCar which is MyPorscheCarrera in this case (see visualization in Figure 13). MyPorscheCarrera has four wheels which are FrontLeft, FrontRight, RearLeft and RearRight. The connection between MyPorscheCarrera and FrontLeft respectively FrontRight is an instantiation of the connection between SportsCar and NarrowWheel, whereas the connection between MyPorscheCarrera and RearLeft respectively RearRight is an instantiation of the connection between SportsCar and BroadWheel. Listing 23 contains two instance facet navigations. The first one enables to retrieve the two front wheels which are FrontLeft and FrontRight. The second shows a level cast to the deep type CarType and returns all four wheels of MyPorscheBoxster.

```
context MyPorscheCarrera
self._SportsCar_.front --Bag{FrontLeft, FrontRight}
self._CarType_.wheels -- Bag{FrontLeft, FrontRight, RearLeft, RearRight}
```

Listing 23: Another example for an useful instance facet navigation.

In the following, the application of a type cast and another usage of level cast is presented when conducting an instance facet navigation. Figure 23 shows the association between CarCompany and Car on the ontological level O1. According to this, a CarCompany can produce multiple Cars. In the O2 Car, CarCompany and its connection are instantiated. In this level, Porsche relates to MyPorscheCarrera, MyPorscheBoxster and MyPorscheCayenne.

Assuming the context is Porsche, Listing 24 presents different OCL expressions in a deep OCL dialect using instance facet navigation which are described hereafter. The first expression simply returns all Cars of the 02 produced by Porsche which are MyPorscheCarrera, MyPorscheBoxter and MyPorscheCayenne. In the second expression only those Cars of kind SportsCar are selected which are MyPorscheCarrera and MyPorscheBoxster. The

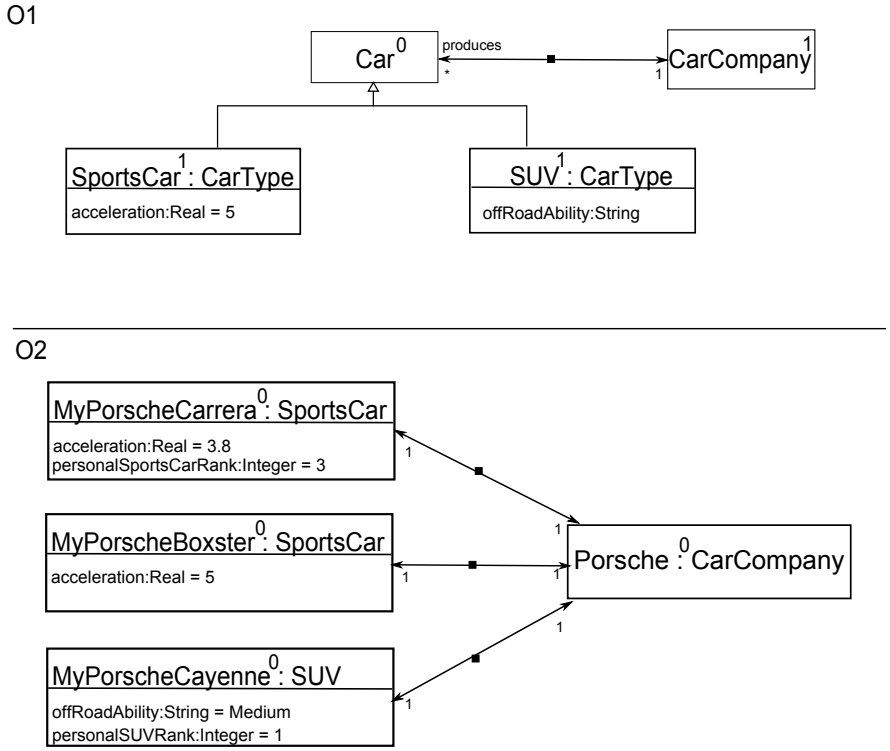

Figure 23: Association between a CarCompany and Cars

third expression accesses the acceleration attribute defined in SportsCar in the O1. However, this is not possible as the static type in this case is Car which does not have an acceleration attribute. Therefore, in the fourth expression a type cast is done which changes the static type to SportsCar and thus enables accessing the acceleration attribute. As the prior select statement ensures that the actual type is SportsCar, this expression delivers a result (and so not OCLInvalid) which is a Bag containing 3.8 and 5. The fifth expression makes use of a select expression which delivers a Bag containing only the specific clabject MyPorscheCayenne. In the next expression, the personalSUVRank attribute is accessed which was defined just in MyPorscheCayenne resided in the O2. This is not valid as the static type Car does not have any attribute called personalSUVRank. In order to overcome this problem, in the last expression a level cast from Car to MyPorscheCayenne is conducted enabling the access to the personalSUVRank attribute. Because the actual clabject resulting from the select statement is MyPorscheCayenne, a result is delivered (and so not OCLInvalid) which is a Bag containing 1. This expression highlights that a level cast not only allows static casting to clabjects which are types or deep types, but also to clabjects which are instances and deep instances of the current clabject.

```
context Porsche
self._CarCompany_.produces -- (1) Bag{MyPorscheCarrera, MyPorscheBoxster,
    MyPorscheCayenne}
self._CarCompany_.produces->select(oclIsKindOf(SportsCar)) -- (2) Bag{
    MyPorscheCarrera, MyPorscheBoxster}
self._CarCompany_.produces->select(oclIsKindOf(SportsCar)).acceleration --
    (3) not possible
self._CarCompany_.produces->select(oclIsKindOf(SportsCar)).oclAsType(
    SportsCar).acceleration -- (4) Bag{3.8, 5}
self._CarCompany_.produces->select(a|a=MyPorscheCayenne) -- (5) Bag{
    MyPorscheCayenne}
self._CarCompany_.produces->select(a|a=MyPorscheCayenne).personalSUVRank --
    (6) not possible
self._CarCompany_.produces->select(a|a=MyPorscheCayenne)._MyPorscheCayenne_
    .personalSUVRank -- (7) Bag{1}
```

Listing 24: Different Examples for an instance facet navigation.

3.2.5. Retrieving Attribute Values

In section 3.1.5 the ambiguity when retrieving values of clabject attributes concerning the ontological and linguistic dimension was indicated. Referring to the example in Figure 19, this means that despite the fact that the datatype of the acceleration attribute of the SportsCar entity is defined as Real (ontological definition), the value of a clabject attribute in general is defined as String in the PLM (linguistic definition).

The deep OCL dialect considers this ambiguity and evaluates to the ontological specified datatype when taking in an ontological perspective and to a String when taking in a linguistic perspective.

Listing 25 presents an example where an ontological attribute value is accessed. As the acceleration attribute is defined as Real the expression returns a value of type Real which is 5 in this case.

```
context SportsCar
self.acceleration -- 5
self.acceleration + 2.3 -- 7.3
```

Listing 25: Examples of accessing an ontological attribute value.

A deep OCL dialect also enables to invoke all operations which are specified for the basic primitive datatypes (see Figure 8). Therefore, for instance, the '+' operation for adding another value to the acceleration attribute can be used like this is the case in the second expression of Listing 25.

In Listing 26 the attribute acceleration is accessed via the linguistic dimension. Hereby, the expression evaluates to a String which is in accordance with the specification of an

attribute value in the PLM. Therefore, for example, the '+' operation cannot be invoked but the concat operation which is defined for the String datatype.

```
context SportsCar
self._l_.getAllAttributes()->any(name=''acceleration'').value --  ''5''
self._l_.getAllAttributes()->any(name=''acceleration'').value + 2.3 -- not
    valid
self._l_.getAllAttributes()->any(name=''acceleration'').value.concat(''test
    '') --  ''5test''
```

Listing 26: Examples of accessing a linguistic attribute value.

3.3. Requirements for the Definition and Validation of OCL Constraints in a Deep OCL Dialect

Besides its purpose as a query language for model elements and their properties, another important aspect of OCL is to specify constraints (see also 2.3.1). Constraints are based on OCL expressions and " when the value of an expression is of Boolean type, it may be used as constraint" [39]. OCL enables specifying constraints for model elements (invariants) but also for their operations (pre- and postconditions). With respect to the scope of this thesis, the focus is placed on invariants.

The specification of OCL is mainly designed for the definition and validation of constraints in a two-level modeling environment. Therefore, a specification for a deep OCL dialect has to be elaborated which defines how definition and validation of constraints are interpreted and applied in a deep modeling context.

With respect to Warmer and Kleppe [39], constraints which are set on model elements on the level M_n affect elements on the M_{n-1}. Regarding invariants in particular, the OCL specification [31] states that an invariant is referred to a type and must be true for all instances of this type at any time. Thus, constraints are set on the type level and validated on the instance level in two-level models.

In a deep model, however, there can be more than two levels which means that there is no type and instance level like in the two-level case. Therefore, a deep OCL dialect has to define where in deep models constraints can be specified.

Moreover, the principle of Deep Characterization has to be taken into account when validating constraints in deep models. Deep Characterization refers to the fact that model elements in a deep model not only control instances the level below but also elements further levels below. Hence, the question arises which effect Deep Characterization has on the validation of constraints in a deep modeling environment. With respect to this, a deep OCL dialect has to specify on which respectively over how many levels constraints are checked in deep models.

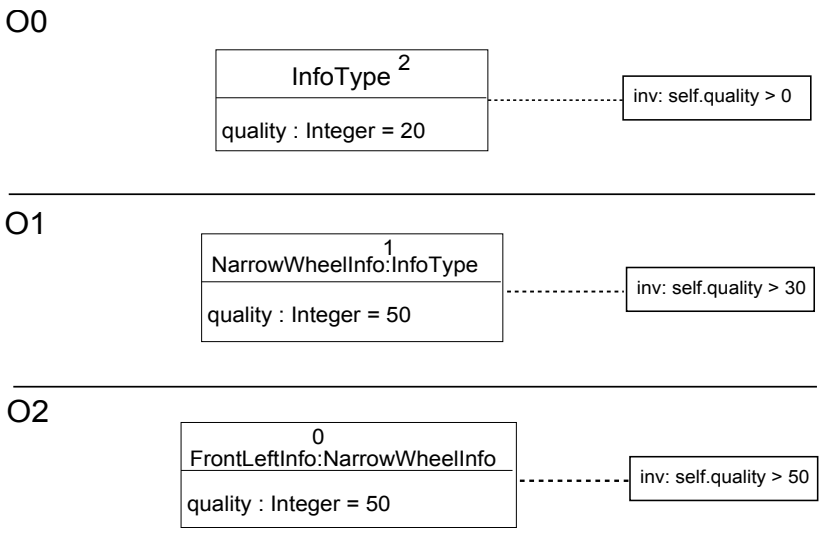

Figure 24: Definitions of constraints in a deep model.

3.4. Proposals for the Definition and Validation of OCL Constraints in a Deep OCL Dialect

This section develops a specification for the definition and validation of constraints in a deep OCL dialect. Thereby, the requirements regarding deep model features which were elaborated in the previous chapter are considered.

3.4.1. Definition of Constraints

In a two-level model, constraints are specified on the type level and validated on the instance level. As a deep model can contain more levels then just a type and instance level, a deep OCL dialect has to clarify where constraints can be defined in a deep model.

Each level of a deep model consists of clabjects which have an instance and type facet. Referring to the two-level based OCL specification [31] which amount that constraints are set for types, it is decided that on every clabject in the model constraints can be defined as clabjects always can be perceived as types. In essence, this means that constraints can be specified on every level for every clabject in a deep OCL dialect.

Figure 24 shows the entity InfoType which is instantiated over two ontological levels. Instances respectively deep instances of InfoType are NarrowWheelInfo and FrontLeftInfo which are also shown here. InfoType has an attribute quality which has a durability of two and therefore is present in instances the direct level below and one further level below. The example illustrates that an invariant can be set for this attribute regardless at which level the holding clabject resides.

In the next section, it is discussed at which levels and over how many levels constraints are validated and how redefinitions of constraints like in Figure 24 are interpreted.

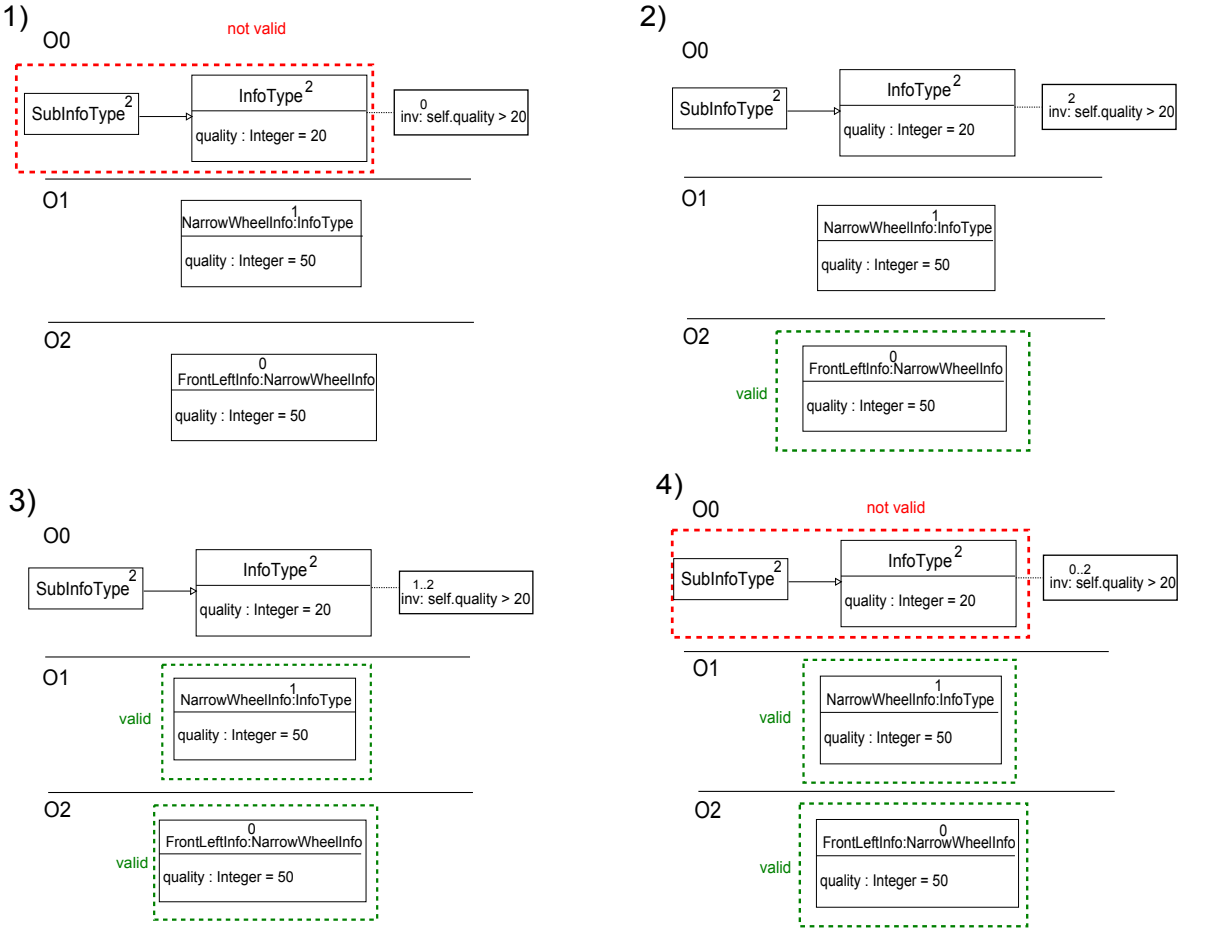

Figure 25: Examples of potency and potency ranges for invariants in a deep OCL dialect.

3.4.2. Validation of Constraints

Constraints in a two-level model are validated on the instances of the instance level. A deep model, however, can consist of more than two levels and because of this there exists no instance level in particular like in the two-level modeling approach.

As clabjects always have an instance facet in addition to the type facet (except those in O0), constraints can be checked on every clabject and thus on every level in a deep OCL dialect.

In deep models, clabjects can be instantiated over multiple levels. Hence, a deep OCL dialect has to specify over how many levels a constraint should be validated. This can be referred to as a *deep validation* of constraints.

A possible approach can be to define potencies or potency ranges for constraints on clabjects. Figure 25 shows the application of this concept with different examples.

In the first example, the potency of the invariant is 0 so that this invariant is only checked for the InfoType and its subtype SubInfoType. As the quality is 20, the invariant is not fulfilled so that this clabject and its subtype are not valid in this case. The second example has an invariant of potency 2 so that this invariant is validated for all deep

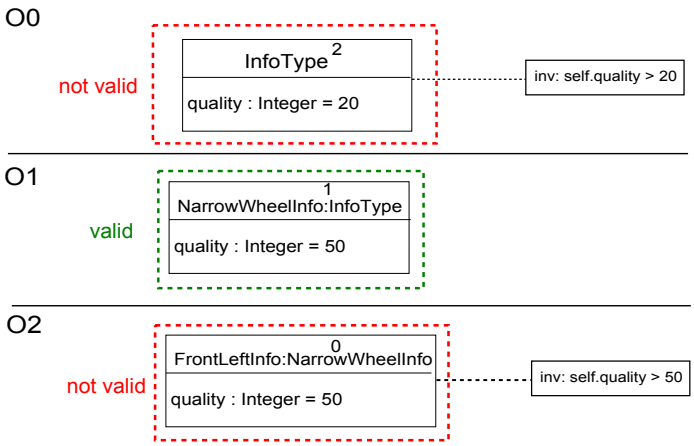

Figure 26: Example of the evaluation of a redefined invariant in a deep OCL dialect.

instances of InfoType two levels below which is in this example FrontLeftInfo. It has a quality of 50 and so the invariant is fulfilled for this clabject. In the third example, a potency range from 1 to 2 is specified for the invariant. This means that the invariant is checked for the instances the level below and the deep instances two levels below. In this case, the invariants are fulfilled and therefore the instantiated clabjects are considered to be valid. The last example uses the complete range so that the invariant is checked for the clabject itself where the invariant was defined and on all its instantiations and deep instantiations.

In general, the potency respectively the upper potency of a constraint must neither exceed the potency of the clabject where it is defined nor the durability of any property of a clabject which is used in the constraint. If no potency or potency range is given, the constraint is validated for the clabject where it is defined and all its instances and deep instances if it has any. This is the same behavior like the example 4 in Figure 25.

The previous section has shown that constraints can be redefined in a deep OCL dialect. With respect to this, it holds that if two constraints concern the same property of a clabject the constraint is validated which was defined on a higher ontological level. This is illustrated in Figure 26. In this case, the invariant for the quality attribute which is set for InfoType is checked for InfoType and NarrowWheelInfo. As it is redefined for FrontLeftInfo, however, it is not checked for this clabject. Instead of this, the redefined invariant is validated.

4. Implementation

The major goal of this thesis was to enrich the Melanee workbench (see section 2.6) with an interactive level-agnostic OCL console which enables live queries on LML models adhering to a deep OCL dialect presented in the previous chapter.

Therefore, an Eclipse plug-in was developed, i.e. *org.melanee.ocl.service*, which defines 2 extensions. These are listed in the plugin.xml file of this plug-in which is shown in Listing 27. The first extension, specified from line 4 to 10, provides the core of Melanee with an OCL service which enables to evaluate OCL expressions based on a deep OCL dialect. The second extension, specified from line 11 to 17, makes use of the first extension and builds the underlying implementation of the aforementioned level-agnostic OCL console.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?eclipse version="3.4"?>
3  <plugin>
4     <extension
5           point="org.melanee.core.constraintlanguage.service">
6        <service
7              class="org.melanee.ocl.service.OCLService"
8              id="org.melanee.ocl.service">
9        </service>
10    </extension>
11    <extension
12          point="org.eclipse.ui.console.consoleFactories">
13       <consoleFactory
14             class="org.melanee.ocl.service.console.consoleFactories.
                  LMLOCLConsoleFactory"
15             label="LML OCL">
16       </consoleFactory>
17    </extension>
18
19 </plugin>
```

Listing 27: Plugin.xml file of the *org.melanee.ocl.service* plug-in.

In the following, first the usage of this console within Melanee is described. After this its implementation is explained. Then the implementation of the underlying OCL service is presented. In the end, the current state of another plug-in, the *org.melanee.validation.service*, is briefly addressed as it also makes use of the OCL service. This plug-in aims to enable the validation of specified constraints in a LML model.

4.1. Usage of the Level-agnostic OCL Console in Melanee

Figure 27 shows a screenshot of an example usage of the level-agnostic OCL console within Melanee.

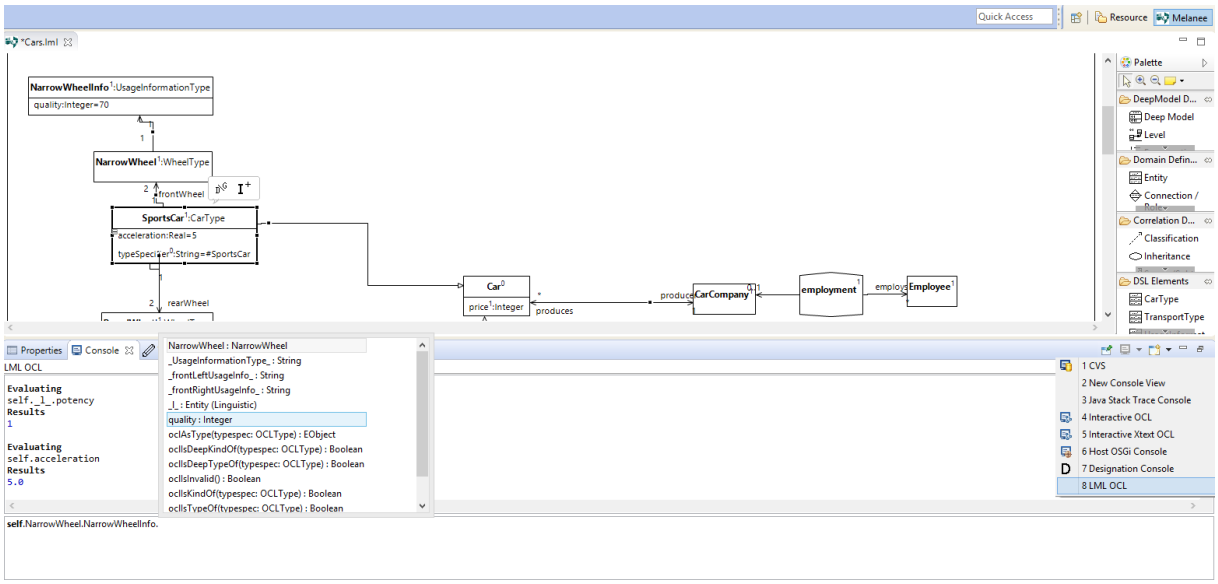

Figure 27: Screenshot of the usage of the level-agnostic OCL console within Melanee.

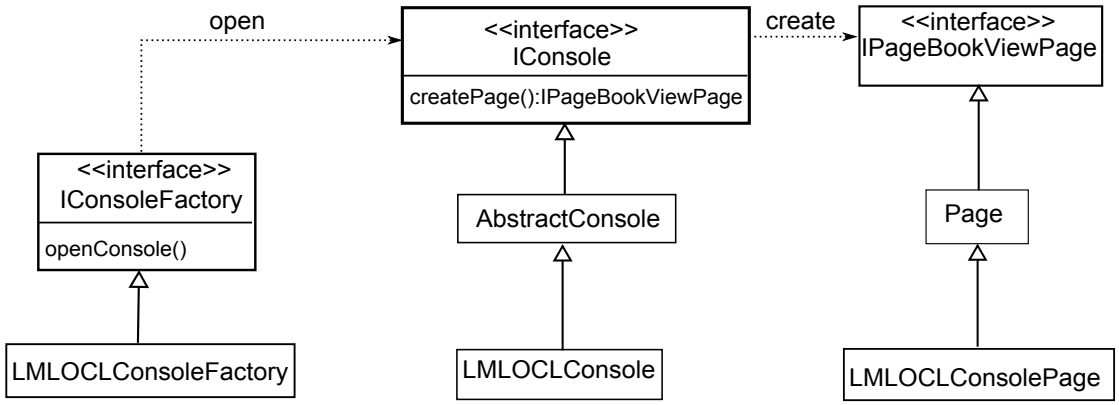

Figure 28: Necessary classes for the level-agnostic OCL console implementation.

The console can be selected from a list of available consoles. In the screenshot, the list is displayed in the bottom right corner. When the console was selected, the console page is shown, which contains an output field at the top and an input field at the bottom. After the selection of a context element in the LML diagram, an OCL expression can be entered in the input field of the console page. In order to help users in the creation of a syntactic and semantic correct expression, a content assistant was implemented which is also shown in the screenshot. The results of the evaluation of an expression are presented in the upper field of the console page, the output field, to the user. A detailed user manual of the level-agnostic OCL console can be found in the Appendix.

4.2. Implementation of the Level-agnostic OCL Console

In order to implement the level-agnostic OCL console, basically three classes are necessary. These and their interplay are visualized in Figure 28.

The starting point for the level-agnostic OCL console implementation is the *LMLOCLConsoleFactory* class which implements the *IConsoleFactory* interface. Implementing this interface, the class serves as a console factory extension by extending the extension point *org.eclipse.ui.console.consoleFactories* (see also Listing 27). This extension is shown in the menu of the console view displayed in the bottom right corner of the screenshot in Figure 27. Moreover, it is responsible for opening a console in the console view by invoking the *openConsole* method.

The console is implemented by the *LMLOCLConsole* class. It extends the *AbstractConsole* class which implements the *IConsole* interface. This interface encompasses a *createPage* method which returns a new page (i.e IPageBookViewPage) for this console. Hereafter, the page is presented in the given console view.

The *LMLOCLConsolePage* class forms a implementation for a console page. It is a subclass of the class *Page* which implements the *IPageBookViewPage* interface. By providing, for example, an ISelectionListener which listens whether a new context element was selected and a InputKeyListener which helps to register when the user is typing in the input field of the console, the LMLOCLConsolePage class manages user interactions. For example, it is recognized when the enter button is pressed which leads to an evaluation of the specified expression whose result is displayed in the output field of the console.

4.3. Implementation of the OCL Service

The *OCLService* class provides an extension for the *org.melanee.core.constraintlanguage.service* extension point, listed in Listing 27. It implements the *IConstraintLanguageService* and thus has to provide two *evaluate* methods. The definitions of these methods in the *IConstraintLanguageService* interface are presented in Listing 28.

```
1  public interface IConstraintLanguageService {
2
3          /**
4           * Evaluates an expression in a constraint language
5           *
6           * @param context the clabject on which the expression is evaluated
7           * @param expression the expression to evaluate
8           *
9           * @return The result of the evaluation
10          */
11         public Object evaluate(Clabject context, String expression) throws
               Exception;
12
13         /**
14          *
15          * @param definitionContext the context on which the expression is
                defined (e.g. if an invariant is defined on a higher level)
16          * @param context the clabject on which the expression is evaluated
17          * @param expression the expression to evaluate
18          *
19          * @return The result of the evaluation
20          */
21         public Object evaluate(Clabject definitionContext, Clabject context
               , String expression) throws Exception;
22
23 }
```

Listing 28: The IConstraintLanguageService interface.

The first evaluate method has two arguments and evaluates an OCL expression on a given contextual clabject. The second one has three arguments and evaluates an OCL expression, which was defined on another definition contextual clabject, on a given contextual clabject. This method is useful, for instance, when a constraint is defined on a higher level clabject than the given contextual clabject.

The evaluation in the aforementioned methods is done by using the API for parsing and evaluating OCL expressions provided by Eclipse OCL Project, which was described in section 2.5.3. The API is presented in Figure 29.

The central element and entry point of the API is the *OCL* class. It encompasses an *Environment* and an *EvaluationEnvironment* which are created by an *EnvironmentFactory*.

The Environment, for example, stores variables which are created during the evaluation of an OCL expression and maintains the selected context element. The implemented *LMLEnvironment* class is an environment adapted to the features of LML. It extends the *AbstractEnvironment* class which implements the Environment interface.

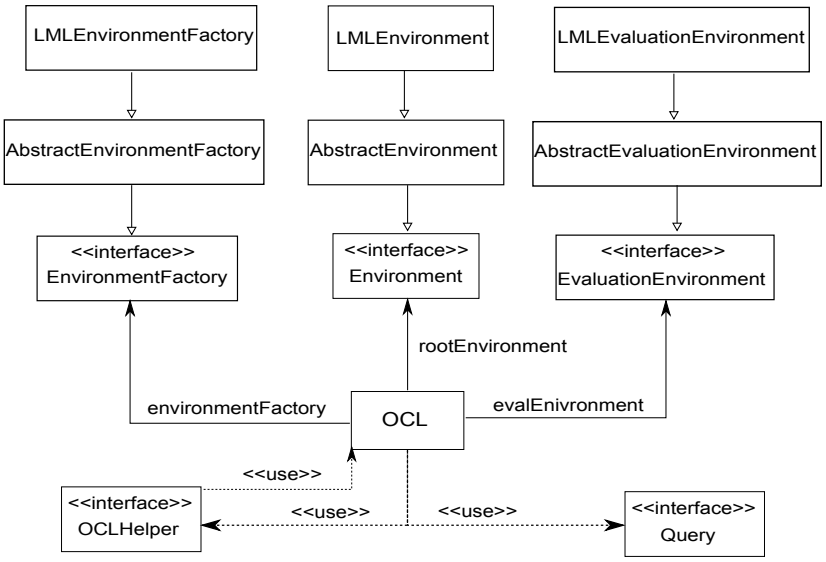

Figure 29: Eclipse OCL API for parsing and evaluating OCL expressions.

The EvaluationEnvironment manages the current values of variables and also enables navigation to property values or associations. The *LMLEvaluationEnvironment* class is a LML specific implementation of the *AbstractEvaluationEnvironment* class which implements the EvaluationEnvironment interface.

The LMLEnvironmentFactory is a concrete implementation of the *AbstractEnvironmentFactory* which implements the EnvironmentFactory. It enables the creation of the described environments, the LMLEnvironment and the LMLEvaluationEnvironment.

The *OCLHelper* API is mainly designed for parsing OCL expressions whereas the intention of the *Query* API is to evaluate these parsed OCL expressions. Listing 29 indicates how the parsing and evaluation of an expression is done in the evaluate method of the OCLService class.

```
1  public class OCLService implements IConstraintLanguageService {
2          private OCL<EPackage, EObject, EObject, EObject, Enumeration, EObject,
               EObject, CallOperationAction, SendSignalAction, Constraint, EObject,
               EObject> ocl;
3          OCLHelper<EObject, EObject, EObject, Constraint> helper;
4          /**
5          * Constructor
6          */
7          public OCLService(){
8           ocl = OCL.newInstance(new LMLEnvironmentFactory());
9           helper = ocl.createOCLHelper();
10         }
11         @Override
12         public Object evaluate(Clabject context, String expression) throws
               Exception {
13          return evaluate(null, context, expression);
```

```
14          }
15          @Override
16          public Object evaluate(Clabject definitionContext, Clabject context
                , String expression) throws Exception {
17           ...
18           helper.setContext(context);
19           OCLExpression<EObject> parsed = helper.createQuery(expression);
20           Query<EObject, EObject, EObject> query = ocl.createQuery(parsed);
21           return query.evaluate(context);
22          }
23  }
```

Listing 29: Parsing and evaluation of OCL expressions in the evaluate method of the OCLService class.

At first, an OCL environment is produced in the constructor of the class in line 11 which is suitable for parsing and evaluating OCL expressions on a LML model. In the next line, a helper object is created. This helper object is needed in the evaluate method in order to parse the OCL expression. After the contextual clabject for the helper is set in line 21, the given expression is parsed in line 22. Hereby, the createQuery method of the helper can throw a ParserExeption in case of syntactic or semantic errors in the expression. If the parsing succeeds, an OCLExpression is returned. As described in section 2.3.2, this is the basic class for all OCL expressions like visualized in Figure 5. During the implementation, all of these expressions which are represented by abstract classes (e.g. IFExp, PropertyCallExp, IteratorExp etc.) in the Eclipse OCL API as well as OCL types (e.g. AnyType, SetType etc.) visualized in Figure 4 were extended and adapted to the deep modeling environment so that they support the linguistic and ontological dimension and are multi-level aware. Afterwards, a query object is built in line 23 by passing the parsed OCLExpression. In line 24, the query object evaluates the OCLExpression on the contextual clabject by utilizing internally the LMLEvaluationEnvironment and the result is finally returned.

4.4. The Deep OCL Validation Service

The deep OCL validation service is another plug-in for Melanee which allows the validation of constraints in a LML model. It makes use of the org.melanee.core.constraintlanguage.service extension point by using the evaluate method of the IConstraintLanguageService interface which has three arguments. As constraints are based on OCL expressions, the evaluate method is used here to evaluate the result of the expression on a given contextual clabject and thus checks if the constraint is fulfilled. The other contextual clabject which is passed in the method is the clabject for which the constraint was defined.

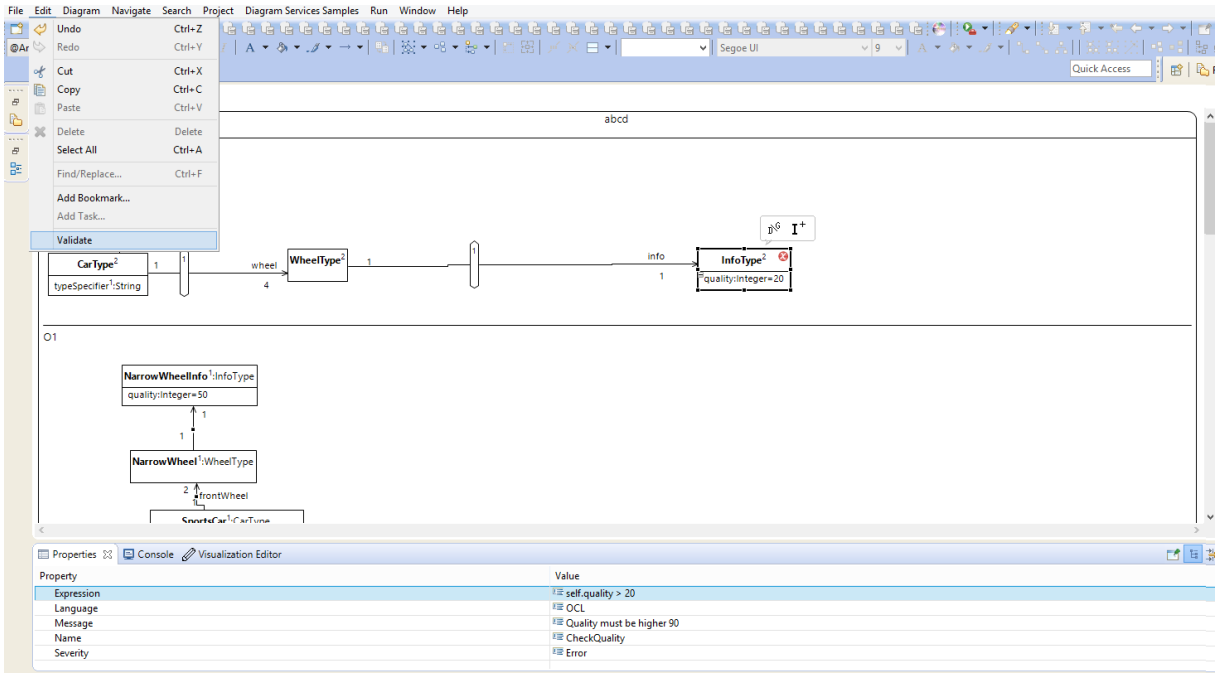

Figure 30: Example usage of the deep OCL validation service in Melanee.

The work on this plug-in is still in progress and the proposals made in section 3.4 has to be taken into account in the implementation. However, the plug-in already offers some functionality so that invariants can be defined for clabjects which are validated on them and its instances as well as its deep instances.

Figure 30 shows a screenshot displaying how invariants currently can be set for clabjects and how they can be validated in Melanee. In this example, the invariant “self.quality>20” is set for the entity InfoType. The definition of the invariant is done in the property view which is placed at the bottom in this example. In order to validate invariants of the LML diagram, the edit menu has to be opened which can be found in the upper left corner in Figure 30. In this menu, the menu point validate can be clicked invoking the validation of invariants. If clabjects are not valid, this is visualized by a red sign in the according clabject visualization. For instance, the InfoType clabject is not valid because its quality is not greater than 20. Nevertheless, the NarrowWheelInfo clabject is a valid instance of InfoType because the quality is 50 and thus greater than 20.

5. Evaluation of the Level-Agnostic OCL Console

This chapter presents the evaluation of the implemented level-agnostic OCL console. The evaluation is done with the help of the well known *Royal and Loyal* example introduced by Warmer and Kleppe [39] which is used by them to explain the OCL.

The ontology of this example only consists of one level and because of this the ontology is extended to three ontological levels in order to have a deep model available for the evaluation. The whole ontology, modeled in LML with the help of Melanee, is shown in Figure 31, whereas the original Royal and Loyal example resides in the O1. In comparison to the original example, the utility class Date, the enumeration Color, all attributes of type Date or Color and all operations are omitted. Furthermore, the role name levels has no ordered annotation as this is not supported in Melanee yet. Due to the lack of space regarding the name of clabjects, the instance of relationships between clabjects in the O1 and the ones in O0 are visualized by dotted lines and not by a ':' after the clabject name.

In chapter 2 of [39], OCL constructs like invariants, pre- and postconditions, initial values etc. are explained with the help of example expressions referring to the Royal and Loyal model. The majority of these expressions are entered in the implemented level-agnostic OCL console while referring to clabjects of the LML diagram in Figure 31 as context.

Taking in the two-level viewpoint of the original model, all of the example expressions chosen by Warmer and Kleppe [39] are intended to be evaluated on the instances of the contextual types where they are defined. Because of this when querying these expressions with the help of the level-agnostic OCL console, in most of the cases the context is set to clabjects which are concrete instantiations, resided in the O2, of the original contexts specified in the book.

Taking in a multi-level perspective, however, the deep OCL dialect supports the evaluation of OCL expressions also on the type facet of clabjects as described in 3.2.1. Therefore, in some examples the context of the expression evaluation is also set to the original context given in the book. The main focus of the Royal and Loyal example lies on association between model elements and how navigation is done via OCL which provides an ideal way to test the concept of navigation in a deep OCL dialect (see section 3.2.4) with the help of the level-agnostic OCL console.

As the adapted expressions created by Warmer and Kleppe do not cover a full evaluation of the deep OCL dialect, additional expressions where entered in the level-agnostic console in order to analyze other deep OCL characteristics.

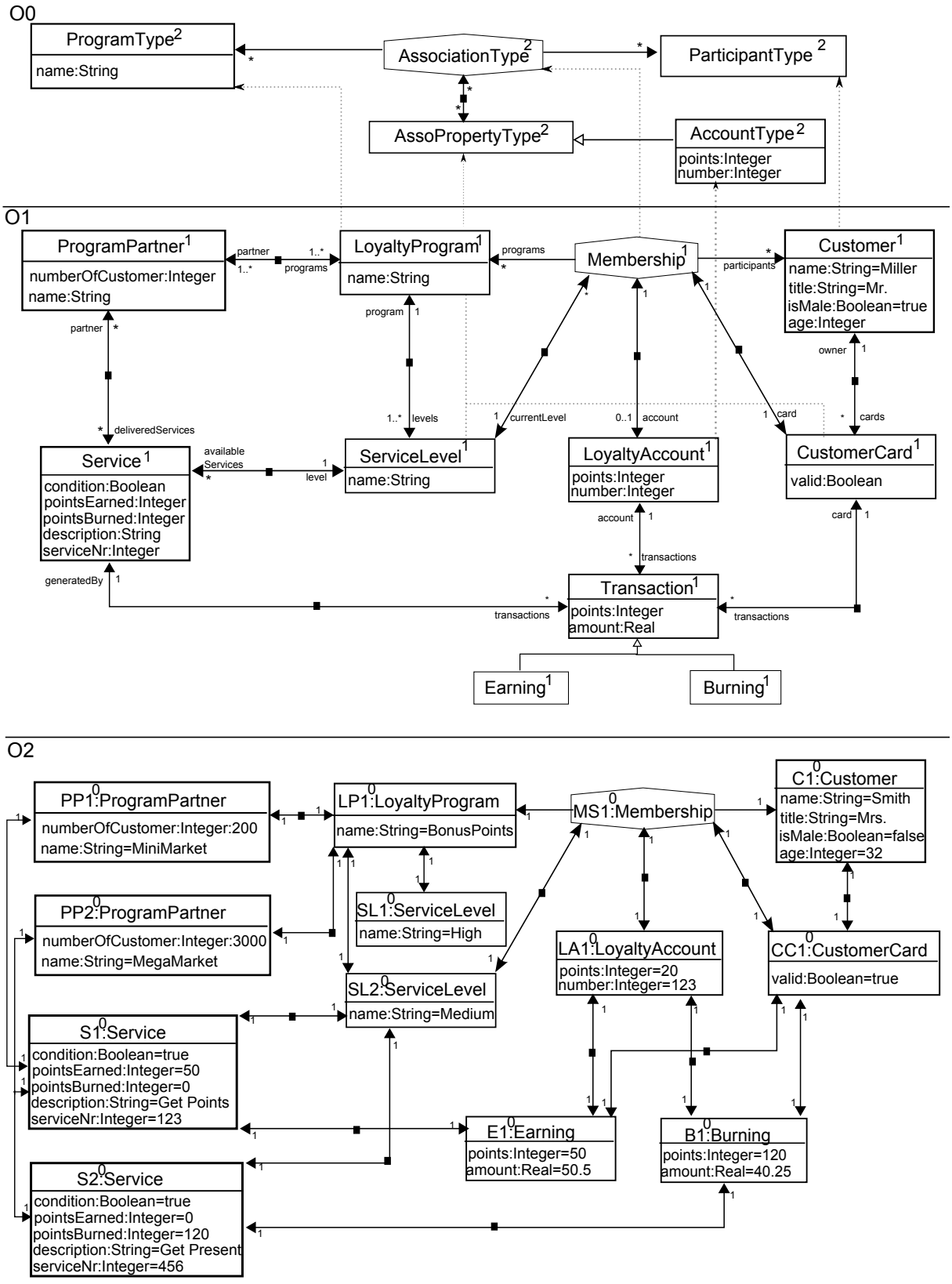

Figure 31: The Royal and Loyal example [39] extended to three ontological levels.

5.1. Evaluation of Example Expressions

In this section, example OCL expressions are queried on clabjects of the extended Royal and Loyal example, visualized in Figure 31, with the help of the level-agnostic OCL console. Their results are interpreted regarding the capabilities and peculiarities of the deep OCL dialect which are discussed in chapter 3. At first, expressions adapted from Warmer and Kleppe [39] are used. In the second part, additional expressions are created and evaluated.

5.1.1. Adapted Expressions

The first expression inspired by the book which is entered in the input field of the implemented console is listed in Listing 30. The expression concatenates the title and name of a Customer instance navigated from a CustomerCard instance. The context is set to the clabject CC1, an instance of the CustomerCard located in O1. As a navigation is performed and thus an instance facet navigation (see 3.2.4) needs to be done, the original expression is slightly modified by applying a level cast from CC1 to CustomerCard. The navigation results in C1 and the whole expression evaluates to "Mrs. Smith" which is the expected result. In this example, the concat method of the primitive type String is successfully used.

```
context CC1
_CustomerCard_.owner.title.concat(' ').concat(_CustomerCard_.owner.name) --
    ''Mrs. Smith''
```

Listing 30: Concatenate owner information of CC1.

As the Customer clabject in O1 already holds values for the name and title attributes, the evaluation of the previous discussed OCL expression can also be done in the context of the CustomerCard. This is shown in Listing 31. In contrast to the previous expression, in this expression a type facet navigation is performed. Executing the query, the correct result in the output field of the console, which is "Mr. Miller", is displayed. This also shows that the direct evaluation of OCL expressions on contextual clabjects enables the access to values of attributes which are just defined for a clabject like name and title in this case.

```
context CustomerCard
owner.title.concat(' ').concat(owner.name) -- ''Mr. Miller''
```

Listing 31: Concatenate owner information of CustomerCard.

Listing 32 presents the next expression which is evaluated within the level-agnostic OCL console. The original expression created by Warmer et al. [39] is used to define the body of an operation for instances of a LoyaltyProgram. Hence, the context is set to LP1 which is an actual instance of the LoyaltyProgram. In the expression, a navigation from

a LoyaltyProgram over its ProgramPartners to their Services is conducted. In order to perform the navigation from LP1, an instance facet navigation is done with the help of a level cast to LoyaltyProgram. The navigation in this example results in a Bag of two S1 and one S2. Applying the asSet operation works and transforms the collection from a Bag to a Set, containing one S1 and one S2.

```
context LP1
_LoyaltyProgram_.partners.deliveredServices->asSet() -- Set{S1,S2}
```

Listing 32: Get all services of partners for LP1.

In contrast to Listing 32, in Listing 33 the context is set to LoyaltyProgram and the original expression without a level cast is queried in the console's input field. In this case, a type facet navigation is perfomed. The result of the navigation is again a Bag and the result of the whole query is a Set, including the Service clabject.

```
context LoyaltyProgram
partners.deliveredServices->asSet() --Set{Service}
```

Listing 33: Get all services of partners for LoyaltyProgram.

The next expression is used in the book to define a body of another operation for instances of LoyaltyProgram. Moreover, an argument of the type ProgramPartner is passed to define the operation body. In order to evaluate the expression within the implemented console, the context is set to LP1 and the argument of type ProgramPartner is represented by PP2 which is an instance of ProgramPartner. The expression itself is an if-then-else expression and is shown in Listing 34. The original expression of the book should deliver a Set of services provided by the passed ProgramPartner if it is included in the contextual LoyaltyProgram. If it is not included, the result should be an empty Set. As an instance facet navigation is conducted here, however, the result type is expected to be a Bag and not a Set, like it is specified in 3.2.4. After typing the expression in the console's input field and evaluating it, the output of the console presents a Bag including S1 and S2 which is the expected result of this example. Additionally, the if-then-else expression and also the includes operation for collections is successfully applied here.

```
context LP1
if _LoyaltyProgram_.partners->includes(PP2)
then PP2._ProgramPartner_.deliveredServices
else Bag{}
endif -- Bag{S1,S2}
```

Listing 34: Get services of a partner if it is included in LP1.

In the next example in Listing 35, the context is an instance of LoyaltyAccount which is LA1 here. The expression collects the values of the attribute amount of all transactions of LA1 and builds its sum. The iteration operation collect is implicitly and the collection

operation sum is explicitly used here. Querying this expression with the level-agnostic OCL console results in 90.75 which is the expected result. Furthermore, the example shows that the amount attribute is correctly treated as a Real type.

```
context LA1
_LoyaltyAccount_.transactions.amount->sum()   -- 90.75
```

Listing 35: Add up the amount of the transactions of LP1.

Another expression inspired by Warmer and Kleppe [39] is shown in Listing 36. It should return all available services for a specific level, selected by a given level name, of an instance of LoyaltyProgram. The level name "Medium" is chosen and the context is set to LP1. The output field of the console presents the correct result which is a Bag including S1 and S2. Moreover, in this example the iteration operation select is utilized which works as expected.

```
context LP1
_LoyaltyProgram_.levels->select(name=''Medium'').availableServices -- Bag{
    S1,S2}
```

Listing 36: Get services of available services of LP1 by selecting level name.

Listing 37 shows an OCL expression which checks if the age of the Customer instance C1 is greater than or equal to 18. After evaluating this expression, the console displays the correct result which is true in this case as the age of C1 is 32. The age attribute is correctly interpreted as an Integer and also the >= operation for the Integer type works properly.

```
context C1
age >=18 -- true
```

Listing 37: Check age of C1.

The expression in Listing 38 is quite similar to the one of the previous example but this time the context is set to an instance of CustomerCard which is CC1 in this example. The expression checks if the owner of the card is 18 or older than 18. A prerequisite for this check is that the navigation results in a single clabject and not in a collection. In this case, like specified in 3.2.4, the navigation results in a single clabject which is C1 here because the multiplicity of the connection between CustomerCard and Customer is one. The whole expression evaluates to true as the owner of CC1, who is C1, is 32.

```
context CC1
_CustomerCard_.owner.age >=18 -- true
```

Listing 38: Check age of owner of CC1.

In the next expression presented in Listing 39 it is verified if the levels of LP1 are including all the levels of the memberships where LP1 is involved. In this example, two

navigations are used which are presented in the second and the third expression of the Listing. The second navigation results in a Bag including SL1 as well as SL2 and the third one results in a Bag containing only SL2. The result of the first expression shown in the console's output is true which is the expected result as the Bag containing SL1 and SL2 includes SL2 which is the only clabject in the bag resulted from the third expression. This example shows a successful application of a navigation over a connection adhering to the deep OCL dialect and the use of the collection operation includesAll within the level-agnostic OCL console.

```
context LP1
_LoyaltyProgram_.levels->includesAll(_LoyaltyProgram_.Membership.
    currentLevel) -- true
_LoyaltyProgram_.levels -- Bag{SL1,SL2}
_LoyaltyProgram_.Membership.currentLevel -- Bag{SL2}
```

Listing 39: Check if levels of LP1 are including all levels of memberships where LP1 is involved.

Listing 40 presents an expression where the contextual clabject is the connection MS1, an instance of Membership. In this example two kinds of navigations are conducted. One from the connection to one of its participants ("_Membership_.participants") and one to another clabject over another connection ("_Membership_.card"). The expression should check if the cards of the participants of MS1 include the card of MS1. The result visualized in the console's output field is true which is the expected value. Hence, navigation from a connection works properly with the help of the level-agnostic console in this example.

```
context MS1
_Membership_.participants.cards->includes(_Membership_.card) -- true
```

Listing 40: Check if cards of participants of MS1 include the card of MS1.

The expression in Listing 41 should return the size of the delivered services of partners of LP1. As described already before in this section, the result of the navigation over partners to delivered services is a Bag containing two S1 clabjects and one S2 clabject. The result of the size operation is 3 which is the correct value in this case.

```
context LP1
_LoyaltyProgram_.partners.deliveredServices->size()
```

Listing 41: Get size of services of partners of LP1.

The next expression shown in Listing 42 is a bit more extensive. In essence, it checks that when all delivered services of partners of LP1 have neither points earned nor points burned, then this implies that its membership must have no account. In this example, three different OCL operations are tested in the implemented console which are the iteration operation forAll, the collection operation isEmpty and the operation implies for

the type Boolean. All of these operations are successfully applied here. The result of the full expression is true which is the expected result as both services resulting from the navigation, S1 and S2, contain either points earned or points burned.

```
context LP1
_LoyaltyProgram_.partners.deliveredServices->forAll(pointsEarned=0 and
    pointsBurned=0) implies _LoyaltyProgram_.Membership.account->isEmpty()
    -- true
```

Listing 42: Check that membership has no account when services of partners of LP1 have no earned and burned points.

The next example expression inspired by Warmer and Kleppe [39] shown in Listing 43 cannot be executed within the console because it invokes the *first* operation on a Bag which is not possible as it is not ordered. In the original Royal and Loyal example the navigation to levels from a LoyaltyProgram is meant to result in an ordered collection indicated by the ordered annotation. Concerning the LML diagram in Figure 31 , however, the result of the navigation in this example is a Bag and not a Sequence as the ordered annotation is not supported yet in Melanee.

```
context LP1
_LoyaltyProgram_.levels->first().name = ''Silver'' -- throws
    ParserException
```

Listing 43: Calling the *first* operation for collections.

In Listing 44, the context is set to an instance of ProgramPartner which is PP2 in this example. In the first expression, the points of all transactions created by the services of PP2 are accumulated without regarding the type of the transaction. The level-agnostic console presents 170 as result in its output field, which is the correct sum of the transactions E1 and B1. In contrast to the first expression, the second expression in Listing 44 selects those transactions which are instances of the clabject Earning. In this case, a select statement in combination with the oclIsTypeOf operation is successfully applied. The result is 50 which is correct as this is the amount of points of the only instance of Earning which is E1.

```
context PP2
_ProgramPartner_.deliveredServices.transactions.points->sum() -- 80
_ProgramPartner_.deliveredServices.transactions->select(oclIsTypeOf(Earning
    )).points->sum() --50
```

Listing 44: Get points for transactions of services delivered by PP2.

5.1.2. Additional Expressions

In deep models there exists another dimension in addition to the ontological one, the linguistic dimension. This dimension respectively its properties can be accessed in a deep

OCL dialect (see section 3.2.2). Listing 45 shows two expressions accessing linguistic properties of the clabject Service. These expressions are queried within the level-agnostic OCL console and the correct results are returned. The result of the first expression is 1 which is the actual potency of Service. The second query returns all linguistic attributes of Service.

```
context Service
_l_.potency -- 1
_l_.getAllAttributes() -- Set{pointsBurned, description, pointsEarned,
    serviceNr, condition}
```

Listing 45: Accessing linguistic properties.

A deep OCL dialect enables type checking over more than one level which is described in section 3.2.3. In Listing 46 the context is set to LA1. The first expression checks if the deep type of LA1 is AccountType, the second expression examines if the deep kind of LA1 is AssoPropertyType. Listing 47 presents the same expressions but this time the context is ServiceLevel. Both expressions are queried on both contexts within the implemented console and the expected results are delivered. Both, LA1 and SL1, are a deep kind of AssoPropertyType, but only LA1 has the deep type AccountType.

```
context LA1
oclIsDeepTypeOf(AccountType) -- true
oclIsDeepKindOf(AssoPropertyType) --true
```

Listing 46: Deep type checking in the deep OCL dialect.

```
context SL1
oclIsDeepTypeOf(AccountType) -- false
oclIsDeepKindOf(AssoPropertyType) --true
```

Listing 47: Deep type checking in the deep OCL dialect.

Listing 48 shows different OCL expressions which are evaluated on the context MS1 with the help of the level-agnostic OCL console. In the first expression, a level cast to the deep type AssociationType of MS1 is conducted. Then, a navigation to AssociationPropertyType is done. Executing this expression, delivers all associated clabjects of MS1 which are of the deep type AssociationPropertyType. In the second expression, the first expression is extended by selecting only those clabjects which are of the deep type AccountType. In this case, a Bag containing only the clabject LA1 is returned which is the expected result. In the next expression the value of the attribute points is tried to access. Querying the expression within the console throws a ParserException which is a correct behavior as clabjects in the resulting Bag are of the static type AssoPropertyType which has no attribute points. In order to overcome this problem, a deep type cast is executed in the last expression. By doing this, the value of the points attribute can be fetched. Finally, the correct value of 20 is shown in the output field of the console.

```
context MS1
_AssociationType_.AssoPropertyType -- Bag{CC1, LA1, SL1}
_AssociationType_.AssoPropertyType->select(oclIsDeepTypeOf(AccountType)) --
    Bag{LA1}
_AssociationType_.AssoPropertyType->select(oclIsDeepTypeOf(AccountType)).
    points -- throws ParserException
_AssociationType_.AssoPropertyType->select(oclIsDeepTypeOf(AccountType)).
    oclAsDeepType(AccountType).points -- Bag{20}
```

Listing 48: Deep type cast in the deep OCL dialect.

In deep models, clabjects can have instances as well as deep instances. Therefore, in the deep OCL dialect the static operation allDeepInstances for clabjects is introduced besides the standard OCL operation allInstances (see section 3.2.3). Listing 49 shows two example expressions of this kind which are evaluated within the level-agnostic OCL console. Both expressions work properly in this case as the first expression results in all instances of AssoPropertyType in O1 and the second expression results in all deep instances of AssoPropertyType which are all located in O2 in this example. The third expression results in the same Set of clabjects like in the second expression because all deep instances of AssoPropertyType have a degree of one in this example since there are only 3 ontological levels.

```
AssoPropertyType::allInstances() -- Set{LoyaltyAccount, ServiceLevel,
    CustomerCard}
AssoPropertyType::allDeepInstances() -- Set{LA1, SL2, SL1, CC1}
AssoPropertyType::allDeepInstances(1) -- Set{LA1, SL2, SL1, CC1}
```

Listing 49: Querying instances of a clabject in the deep OCL dialect.

Listing 50 presents an example where the allInstances operation is invoked on a linguistic meta-type which is Entity in this case. The evaluation of this expression within the console delivers all entities of the ontology regardless in which ontological level they are placed. This is the expected result of the query.

```
Entity::allInstances() -- Set{ProgramPartner, LP1, AssoPropertyType ... }
```

Listing 50: Querying instances of the linguistic meta-type Entity.

6. Future Work

In this thesis, a deep OCL dialect was specified and a level-agnostic OCL console for querying OCL expressions in LML diagrams as well as a deep OCL validation service for the definition and validation of constraints were implemented, based upon this dialect. However, there are still missing parts in both implementations which reveal the potential for future work. These parts are described in the following.

6.1. Query of OCL Expressions in LML Diagrams

The focus in this thesis was primary set on accessing attribute or navigation values, when querying properties of clabjects in a deep OCL dialect. In the future, also queries of operations respectively their body should be taken into account.

In the PLM, operations are called *methods* so this term will be used in the following. The body of a method can be set via the property view in Melanee and is currently defined as String in the PLM. When querying the method in the level-agnostic OCL console, only the String is returned without any further interpretation of it. The body of a method, however, can be compared to the body expression in OCL which defines the result of an operation. According to the OCL specification " the expression must conform to the result type of the operation" [31]. As long as methods in Melanee have no datatype, this conformance check is not necessary. Nevertheless, when accessing a method of a clabject, the body should be interpreted as an OCL expression in the future so that the result of the query is the result of the expression defined in the method body.

In the LML particpations of connections cannot be annotated as *ordered* yet. Because of this, navigations executed in the level-agnostic OCL console never result in an *OrderedSet* or a *Sequence* at the moment. Consequently, the specific operations of these collection types (e.g. first, last, append, prepend) can also not be used. Therefore, expanding the characteristics of participations so that they can be annotated as ordered would enable to make use of the OCL collection types of OrderedSets as well as Sequences and its operations.

6.2. Definition and Validation of Constraints in LML Diagrams

As indicated in 4.4, the work on the deep OCL validation service is not finished yet. Currently, an invariant can be specified for a clabject and is validated on its instances and its deep instances. The concept of *potency and potency ranges for constraints* described in section 3.4 needs to be implemented in the future in order to determine at which levels the invariant should be evaluated.

In the LML, participations between connections and their participants have multiplicities. These can be regarded as a shorthand form of an OCL constraint. The participation in Figure 13, for instance, with the role name wheel of the connection between CarType and WheelType has a multiplicity of 4. This can also be expressed in OCL like shown in Listing 51.

```
context CarType
inv: self.wheel->size()=4
```

Listing 51: Example for expressing a multiplicity constraint in OCL.

Hence, another feature which could be implemented in Melanee is that multiplicities in LML diagrams are transformed into OCL constraints so that they are validated the levels below. Furthermore, users must be able to assign potencies respectively potency ranges to the *multiplicity constraints* like it is possible in Nivel [1], which is described in chapter 7.2.

Other OCL Constructs

Every OCL construct is based upon OCL expressions. In this thesis a deep OCL dialect was implemented so that these expressions can be used and evaluated. In the current state, the only construct that can be utilized in the deep OCL validation service is the invariant construct. This means that constraints can be defined for clabjects in Melanee.

Nevertheless, OCL offers also the possibility to specify constraints for operations with the help of *pre- and postconditions*. In the context of LML and Melanee, it has to be examined in the future if and how these kinds of constraints for operations can also be supported.

Other OCL constructs which are also not considered yet in the deep OCL validation service are *derivation rules, initial values, definition of new attributes and methods* and *body of methods*. As these constructs are no real constraints, but rather supplement the diagram with additional properties, it is not of particular importance to implement them in the near future for the deep OCL validation service.

7. Related Work

In this chapter three other deep modeling frameworks are presented and a comparison is conducted between them and Melanee regarding their capabilities of setting constraints to models. Furthermore, the analysis of these frameworks should discover which constraint languages are used and how they are meant to be interpreted, adapted and applied in the respective framework under consideration and in deep models in general.

7.1. MetaDepth

MetaDepth is introduced by de Lara and Guerra [13]. It is a tool which supports textual modeling allowing an arbitrary number of ontological levels and the dual instantiation regarding ontological and linguistic dimension. MetaDepth enables the specification and evaluation of *derived attributes* and *constraints across multiple ontological levels*. The constraint construct is defined in the linguistic meta-model. The Epsilon Object Language (EOL), which extends the OCL with imperative constructs such as assignments, is used to define constraints. The authors claim to make the EOL multi-level aware. Concerning this, de Lara et al. [14] list two differences to the standard OCL implementation.

First, a potency can be set for constraints or derived attributes. The potency indicates at which level the constraint or the derived attribute is evaluated. In contrast to two level models, in deep models this potency can be higher than one. Moreover, de Lara et al. [14] state that "our constraints can use methods and attributes of the linguistic meta-model" [14]. Linguistic properties are accessed like ontological ones and if the name of a linguistic and ontological property collides the prefix '^' is taken in order to refer to the linguistic one.

In contrast to the approach presented in this thesis, a constraint in MetaDepth can only be evaluated on one ontological level but not on multiple levels which is allowed, for example, by potency ranges for constraints described in 3.4.2. Nevertheless, the definition of derived attributes is possible, which is not supported in Melanee yet.

Furthermore, when invoking linguistic properties of a clabject in the deep OCL dialect elaborated in this thesis, this has to be done explicitly by referring to the "_l_"-property of a clabject. In MetaDepth, however, an explicit invocation of a linguistic property has only to be done in the case of name clashes of linguistic and ontological properties.

As mentioned before, MetaDepth uses EOL which reuses the OCL but the syntax is slightly changed in some parts (e.g. isTypeOF in constrast to oclIsTypeOF). In this thesis, first a deep OCL dialect was elaborated which forms the foundation for the implementation of query OCL expressions and defining constraints in LML models. Concerning the MetaDepth tool, no further general investigation of a potential adaption of EOL expressions regarding deep modeling environments is done. For instance, it is not examined

how operations of the AnyType like isTypeOf can be applied and extended using them in deep models which was examined for OCL in section 3.2.3.

7.2. Nivel

Another multi-level framework is *Nivel* which is presented by Asikainen and Männistö [1]. Models in Nivel are created with the help of the weight constraint rule language (WCRL) which is a general purpose knowledge representation language. Nivel enables to specify so called *cardinality constraints*. These are constraints affecting instances of associations holding values for the cardinality and potency. The potency defines at which meta-level the cardinality constraint applies.

Nevertheless, the authors admit that "Nivel defines no constraint language of its own" [1] and so the construct of cardinality constraints remains the only possibility of adding constraints in Nivel. Albeit the authors suggest that WCRL could be used to formalize constraints, they also reject this suggestion as they think that the potential users of Nivel are not familiar with WCRL. Furthermore, they also claim that OCL not might be suitable for expressing constraints in a knowledge representation language as users also might be not familiar with it.

In the context of Melanee, however, modeling can be done in a graphical editor which allows building LML models which supports the look and feel of the UML. Because of this, modelers creating LML models might be more familiar with OCL which initially was meant to be the "formal language used to describe expressions on UML models" [31].

To sum up, the capabilities of Nivel in defining constraints are limited and no profound elaboration is done of how interpreting and applying a constraint language in deep models. Nevertheless, the concept of cardinality constraints is a useful concept which could be adapted to Melanee when defining multiplicities of participations.

7.3. Cross-Layer Modeler

The *cross-layer modeler* is another deep modeling tool introduced by Demuth et al. [15]. It allows modeling an arbitrary number of ontological levels but it does not follow the mechanism of deep instantiation so that potencies are not known and applied here. In contrast to the other described deep modeling tools respectively frameworks in this section, the cross-layer modeler is a graphical model editor. OCL is used to define constraints.

There are two kinds of constraints, which are *constraint templates* and *fixed constraints*. Fixed constraints "are used to check properties that are equal for all instances of a type" [15]. This kind of constraint could be used when setting constraints for direct instances of model elements. Template constraints "are automatically instantiated to create con-

straints that check properties that are not known in advance" [15]. This kind of constraint could be used to set constraints on deep instances two levels below.

However, it is not clarified how to set constraints for elements more than two levels below. This might be difficult to implement in this tool as deep instantiation is not considered yet. Moreover, as opposed to this thesis, no comprehensive elaboration of the interpretation and application of OCL in deep models is made. Additionally, the focus is only set on how setting constraints on models. No possibility is mentioned that enables the query of model elements and its properties with the OCL which is also a query language and not only a constraint language as mentioned in section 2.3. Especially in a graphical editor a console for OCL expressions, for instance, is a useful textual extension.

8. Conclusion

The LML was created to overcome the limitations of the UML identified by several researchers. It is embedded in the OCA and allows modeling an arbitrary number of ontological levels so that deep modeling is supported. The LML model editor Melanee was developed in order to provide researchers, modelers and software engineers with the possibility to experience the LML. Melanee was extended step by step by various features so that it offers users a comfortable way of building LML models.

However, the existing OCL implementation for Melanee was not multi-level aware and in the available OCL console only linguistic properties could be queried. As a consequence, in this thesis a specification of a deep OCL dialect was elaborated. Requirements for the evaluation of OCL expressions as well as for the definition and validation of constraints concerning deep models were examined. Based on this, proposals for these requirements in a deep OCL dialect were made.

As a next step, the deep OCL dialect was implemented which served as the foundation for the development of the level-agnostic OCL console. The evaluation part of this thesis has proven that the implemented console allows queries of linguistic as well as ontological properties. Moreover, it takes the principles of deep modeling into account.

Additionally, based on the deep OCL dialect, the implementation of a deep OCL validation service has been started which enables modelers to define constraints in LML models. Currently, this service already provides useful functionality.

Nevertheless, the OCL validation service is still in the initial phase so that the current implementation has to be refined in the next development steps. The level-agnostic OCL console is already in a more mature stage but still has some room for improvements. The specified deep OCL dialect is the first result of a comprehensive analysis of the OCL interpretation and application in deep models and needs to be further investigated in future research.

References

[1] Timo Asikainen and Tomi Männistö. Nivel: a metamodelling language with a formal semantics. *Software & Systems Modeling*, 8(4):521–549, 2009.

[2] C. Atkinson, M. Gutheil, and B. Kennel. A Flexible Infrastructure for Multilevel Language Engineering. *Software Engineering, IEEE Transactions on*, 35(6):742–755, Nov 2009.

[3] C. Atkinson and T. Kuhne. Model-Driven Development: A Metamodeling Foundation. *Software, IEEE*, 20(5):36–41, Sept 2003.

[4] Colin Atkinson. Meta-Modeling for Distributed Object Environments. In *Enterprise Distributed Object Computing Workshop [1997]. EDOC'97. Proceedings. First International*, pages 90–101. IEEE, 1997.

[5] Colin Atkinson and Ralph Gerbig. Melanie: Multi-level Modeling and Ontology Engineering Environment. In *Proceedings of the 2Nd International Master Class on Model-Driven Engineering: Modeling Wizards*, MW '12, pages 7:1–7:2, New York, NY, USA, 2012. ACM.

[6] Colin Atkinson, Bastian Kennel, and Björn Goß. The Level-Agnostic Modeling Language. In *Software Language Engineering*, pages 266–275. Springer, 2011.

[7] Colin Atkinson and Thomas Kühne. Reducing accidental complexity in domain models. *Software & Systems Modeling*, 7(3):345–359, 2008.

[8] Colin Atkinson and Thomas Kuehne. The Essence of Multilevel Metamodeling. In Martin Gogolla and Cris Kobryn, editors, *UML 2001- The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 19–33. Springer Berlin Heidelberg, 2001.

[9] Colin Atkinson and Thomas Kühne. Rearchitecting the UML Infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321, October 2002.

[10] Carmen Avila and Yoonsik Cheon. OCL 2.0 Quick Reference. `http://www.cs.utep.edu/cheon/cs3360/project/proj1/ocl-ref-short.pdf`, 2008.

[11] Juan Cadavid, Benoit Baudry, Benoit Combemale, et al. Empirical evaluation of the conjunct use of MOF and OCL. In *Experiences and Empirical Studies in Software Modelling (EESSMod 2011)*, 2011.

[12] Eric Clayberg, Eric. Clayberg, and Dan Rubel. *Eclipse Plug-ins, Third Edition*. Addison-Wesley, 2008.

[13] Juan De Lara and Esther Guerra. Deep Meta-modelling with Metadepth. In *Objects, Models, Components, Patterns*, pages 1–20. Springer, 2010.

[14] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. Model-driven engineering with domain-specific meta-modelling languages. *Software & Systems Modeling*, pages 1–31, 2013.

[15] Andreas Demuth, Roberto E Lopez-Herrejon, and Alexander Egyed. Cross-layer Modeler: A Tool for Flexible Multilevel Modeling with Consistency Checking. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 452–455. ACM, 2011.

[16] Eclipsepedia. Eclipse OCL Project. `http://wiki.eclipse.org/OCL`. [Online; accessed 18-June-2014].

[17] Eclipsepedia. Graphical Modeling Framework. `http://wiki.eclipse.org/Graphical_Modeling_Framework`. [Online; accessed 17-June-2014].

[18] Eclipsepedia. Rich Client Platform Wiki. `http://wiki.eclipse.org/Rich_Client_Platform`. [Online; accessed 29-May-2014].

[19] Eclipse Foundation. Eclipse and Eclipse Foundation. `http://www.eclipse.org/org/`. [Online; accessed 29-May-2014].

[20] Eclipse Foundation. Eclipse Modeling Framework. `http://www.eclipse.org/modeling/emf/`. [Online; accessed 17-June-2014].

[21] Eclipse Foundation. Eclipse Modeling Project. `http://www.eclipse.org/modeling/`. [Online; accessed 17-June-2014].

[22] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.

[23] Ralph Gerbig. The Level-agnostic Modeling Language: Language Specification and Tool Implementation. 2011.

[24] Cesar Gonzalez-Perez and Brian Henderson-Sellers. A powertype-based metamodelling framework. *Software & Systems Modeling*, 5(1):72–90, 2006.

[25] Richard C. Gronback. *Eclipse Modeling Project : A Domain-Specific Language Toolkit*. Eclipse series. Addison-Wesley, Upper Saddle River, N.J., 2009.

[26] Object Management Group. OMG Overview. `http://www.omg.org/`. [Online; accessed 24-May-2014].

[27] Object Management Group. MDA Guide Version 1.0.1. `http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf`, 2003.

[28] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.1. `http://www.omg.org/spec/QVT/1.1/PDF`, 2011.

[29] Object Management Group. OMG Unified Modeling LanguageTM (OMG UML), Infrastructure Version 2.4.1. `http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF`, 2011.

[30] Object Management Group. OMG Meta Object Facility (MOF) Core Specification Version 2.4.1. `http://www.omg.org/spec/MOF/2.4.1/PDF`, 2013.

[31] Object Management Group. Object Constraint Language Version 2.4. `http://www.omg.org/spec/OCL/2.4/PDF`, 2014.

[32] Bastian Kennel. A Unified Framework for Multi-level Modeling. 2012.

[33] Thomas Kuehne and Daniel Schreiber. Can programming be Liberated from the Two-Level Style: Multi-level Programming with DeepJava. In *ACM SIGPLAN Notices*, volume 42, pages 229–244. ACM, 2007.

[34] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse Rich Client Platform,Second Edition*. Addison-Wesley Professional, 2010.

[35] Chair of Software Engineering-University Mannheim. Melanee - The Deep-modeling Domain-specific Language Workbench. `http://melanee.org/`. [Online; accessed 18-June-2014].

[36] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE software*, 20(5):19–25, 2003.

[37] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2013.

[38] Dave Steinberg. *EMF : Eclipse Modeling Framework*. Eclipse series. Addison-Wesley, Upper Saddle River, N.J., 2nd edition, 2008.

[39] Jos B Warmer and Anneke G Kleppe. *The Object Constraint Language: Getting your Models Ready for MDA*. Addison-Wesley Professional, 2003.

A. Level-Agnostic OCL Console User Manual

The level-agnostic OCL console is an Eclipse plug-in which serves as an extension for the Melanee workbench. The tool can be downloaded from http://melanee.org/. Furthermore, an installation guide and a walkthrough showing the creation of a new LML diagram are provided. In the following, the usage of the level-agnostic OCL console in the interplay with a LML model within Melanee is presented.

A.1. Walkthrough: Using the Level-Agnostic OCL Console

Open the console view in case it is not opened yet. Select “Window” (Figure 32, 1.) → “ShowView” (2.) → “Other” (3.) . In the “Show View” dialogue select “Console” (4.) and click on “OK” (5.).

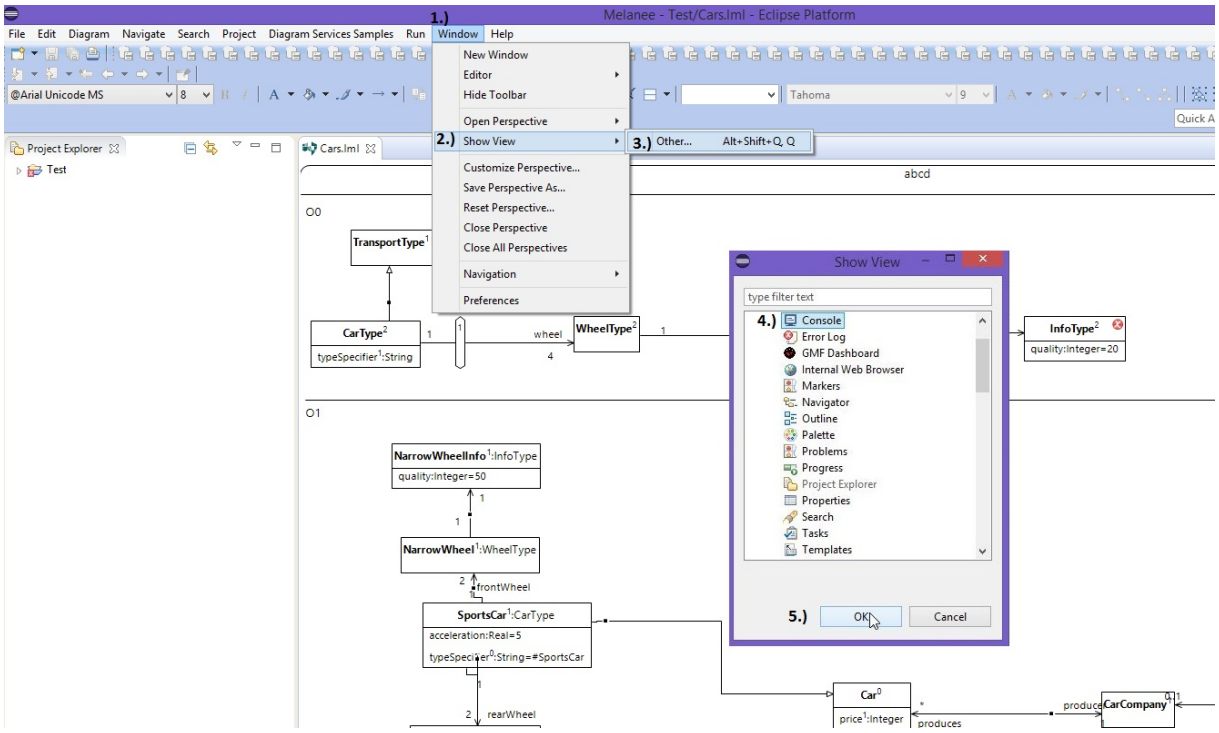

Figure 32: Opening the console view.

Now, in the console view click the “Open Console” button (Figure 33, 1.) and select “LML OCL” (2.) so that the level-agnostic OCL console page will open.

After the console page has opened, select a clabject which should be the context for the OCL query by clicking on it in the LML diagram (Figure 34).

OCL expressions can be entered in the input field of the console which is the lower field of the console page. After each ’.’-operator, ’->’-operator or by pressing “Ctrl” + “Space”, a content assistant is shown (Figure 35). By pressing “Enter”, the specified expression will be evaluated.

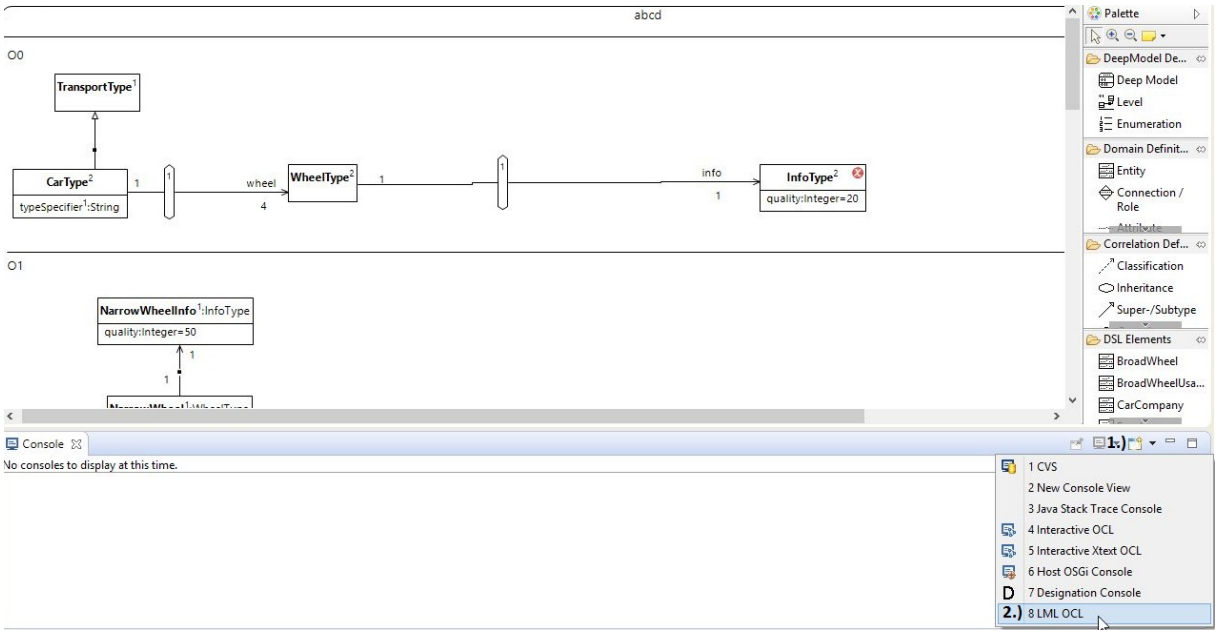

Figure 33: Opening the console view.

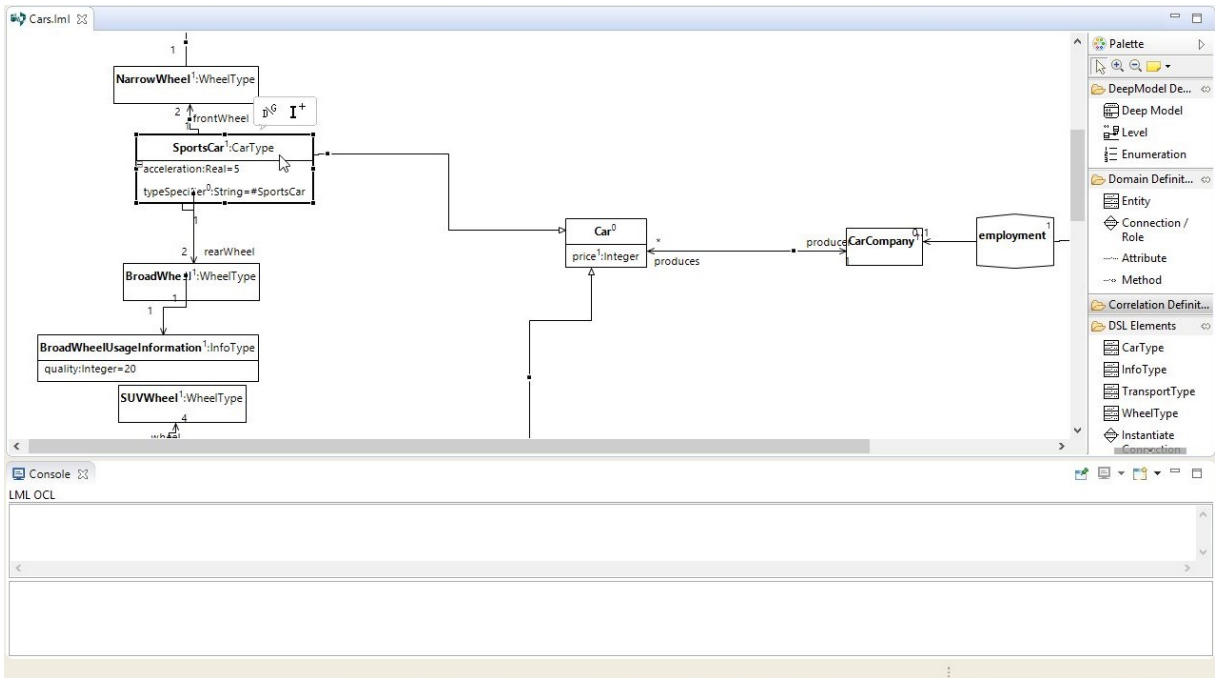

Figure 34: Select contextual clabject in LML diagram.

After the evaluation of the OCL expression, its result is shown in the output field which is the upper field of the console page Figure 36.

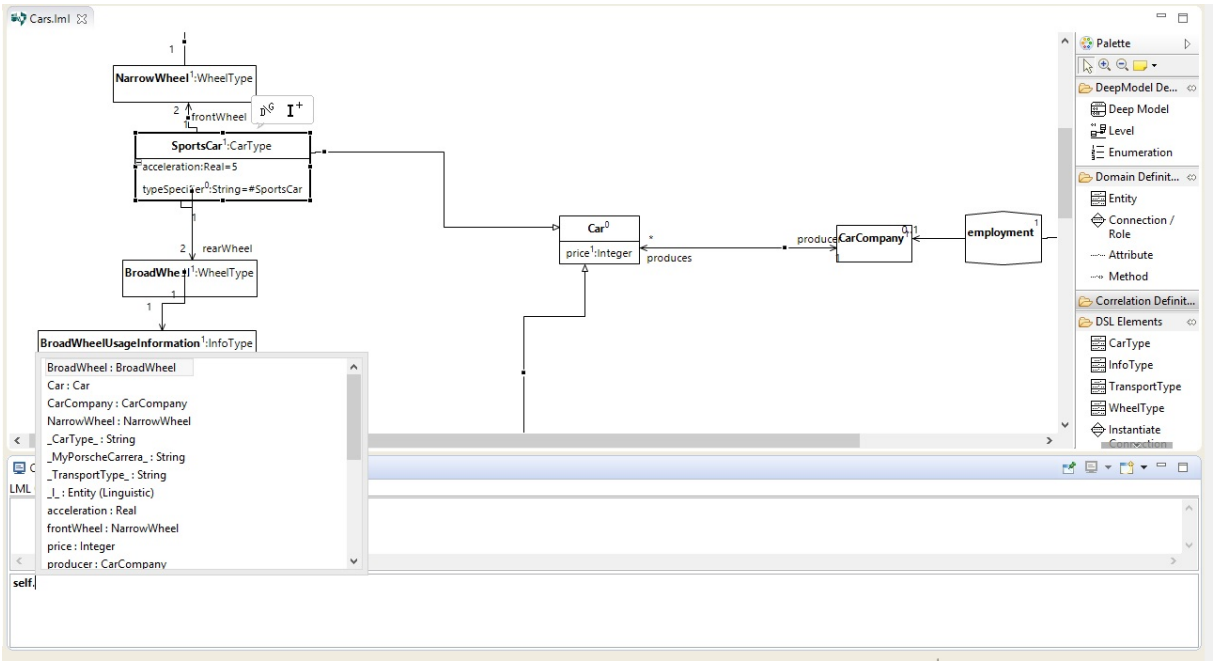

Figure 35: Entering an OCL expression with the help of the content assistant.

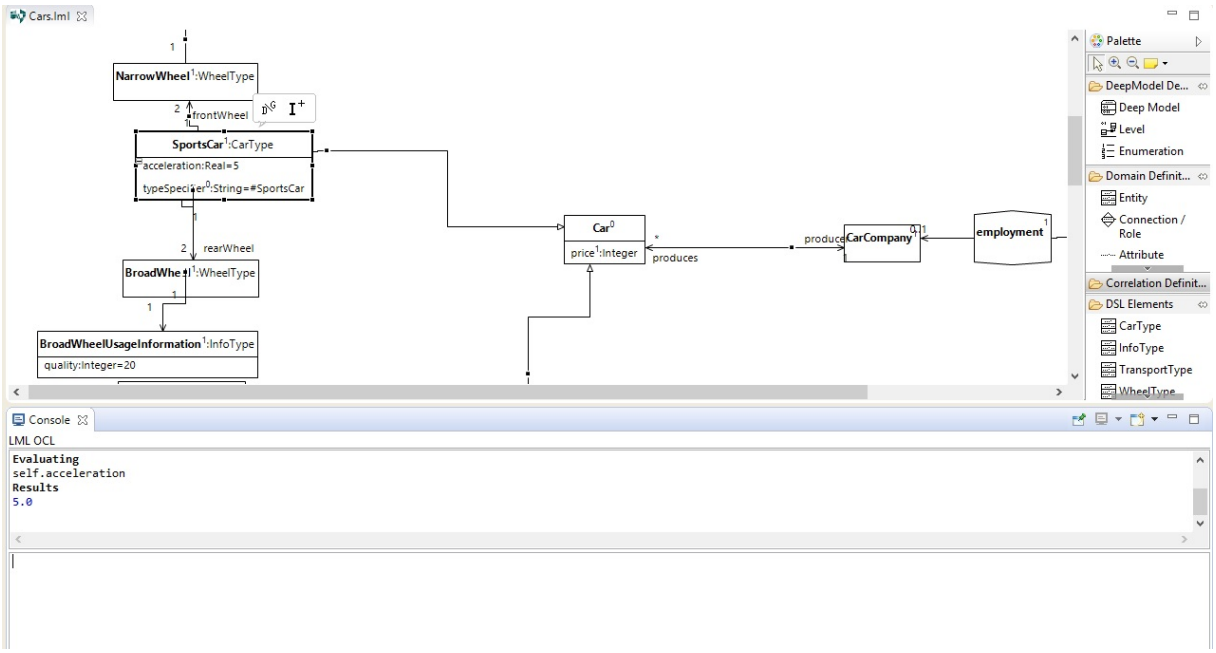

Figure 36: Result of an OCL expression in the console's output field.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mannheim, 13. September 2014 Unterschrift

Abtretungserklärung

Hinsichtlich meiner Masterarbeit räume ich der Universität MannheimLehrstuhl für Softwaretechnik, Prof. Dr. Colin Atkinson, umfassende, ausschließliche unbefristete und unbeschränkte Nutzungsrechte an den entstandenen Arbeitsergebnissen ein.

Die Abtretung umfasst das Recht auf Nutzung der Arbeitsergebnisse in Forschung und Lehre, das Recht der Vervielfältigung, Verbreitung und Übersetzung sowie das Recht zur Bearbeitung und Änderung inklusive Nutzung der dabei entstehenden Ergebnisse, sowie das Recht zur Weiterübertragung auf Dritte.

Solange von mir erstellte Ergebnisse in der ursprünglichen oder in überarbeiteter Form verwendet werden, werde ich nach Maßgabe des Urheberrechts als Co-Autor namentlich genannt. Eine gewerbliche Nutzung ist von dieser Abtretung nicht mit umfasst.

Mannheim, 13. September 2014 Unterschrift