Deep Multi-Style, Multi-Pattern Modeling with DOCL

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

ARNE LANGE

aus Halle (Saale)

Mannheim, 2023

Dekan:	Prof. Dr. Claus Hertling, Universität Mannheim
Referent:	Prof. Dr. Colin Atkinson, Universität Mannheim
Korreferent:	Prof. Dr. Uwe Aßmann, Technische Universität Dresden
Korreferent:	Prof. Dr. Dirk Draheim, Tallinn University of Technology

Tag der mündlichen Prüfung: 15.03.2024

“All models are wrong, but some are useful.”

George Box

Acknowledgements

I want to thank Prof. Dr. Colin Atkinson who guided me along my journey to the Ph.D. He always gave me insightful feedback and challenged me when necessary. The discussions with you were always worthwhile and helped me to make advance my research. Thank you for being involved in my Ph.D process.

I have to thank Prof. Dr. Aßmann and Prof. Dr. Draheim who decided to become my second and third examiners after being presented with the proposed topic of the thesis. Thank you for reading my thesis!

I would like to extend my gratitude to the team of collaborators who have helped me to understand their ideas and approaches in the field of Multi-level modeling. A special thanks to Thomas Kühne for sharing his expertise with me. He brought a unique perspective that enriched every discussion we had.

I thank my colleagues at the University of Mannheim. With them, I had enlightening discussions and gained insight from very different perspectives. In my work as a speaker of the doctoral convent of my faculty, I was able to meet and connect with Ph.D. candidates from different faculties and discuss the trials and tribulations of being a Ph.D. candidate. I also want to thank all the members (also former members) of the doctoral convents for working with me whilst organizing events or working on university political issues.

My parents, who supported me tremendously, are the main reason I even could dream of getting into a PhD. They supported me in every decision leading to this degree and encouraged me to pursue the highest possible education for myself. Thank you for believing in me and for your continuous encouragement during all those years.

Last but not least I want to name a few friends and colleagues during this endeavor that made it fun, these are: Anne, Kilian, Hanna, Valerie, Alisa, Katha, Hannes, Julius, and Sabine.

Abstract

Since the standardization of the UML in the 1990s, object-oriented modeling has assumed growing importance, not only for information systems development and software engineering but also for wider conceptual modeling and ontology applications. The core challenge when modeling is to create models that are sufficiently abstract to leave out detail that is superfluous to the modeling goal, but sufficiently precise to avoid ambiguity about the properties of the domain. For this reason, graphical notations like the UML, which excel in the former, are usually complemented by textual "constraint" languages, such as the OCL, which excel in the latter.

While such language partnerships have been, and still are, used with great success to describe system architectures and designs, they have proven less successful at supporting more challenging applications that require greater flexibility and adaptability without sacrificing abstraction and precision. Examples include domain-specific language design and usage, models that capture complete system life cycles from inception to operation, and ontologically grounded models that have a sound conceptual foundation. Such applications usually require models to go beyond the traditional "two-level" types/instance dichotomy underpinning the UML/OCL partnership and allow model elements to have classifications relationships over three or more levels. However, while there has been significant research into making the graphical component of the aforementioned language partnerships multi-level adjuvant, their constraint language partners have been left behind.

This thesis addresses this problem by presenting a multi-level adjuvant version of OCL to partner with an existing, multi-level-adjuvant dialect of the UML class diagram notation (LML). By adding features to facilitate ontological and linguistic reflection and make constraints multi-level aware, the language supports much more flexible approaches to multi-level modeling, whilst still retaining the precision and model soundness provided by OCL. This is achieved through the notion of multi-level modeling styles and patterns. After presenting DOCL, and showing how it can be used to define a range of flexible deep modeling styles and patterns, the thesis shows the advantage of the developed technology in four, benchmark multi-level modeling challenges.

Zusammenfassung

Seit der Standardisierung der UML in den 1990er Jahren hat die objektorientierte Modellierung, nicht nur für Informationssysteme und Softwaretechnik, sondern auch für breitere Anwendungen in der Konzeptmodellierung und Ontologie, an Bedeutung gewonnen. Deren Hauptziel besteht darin, Modelle zu schaffen, die abstrakt genug sind, um unnötige Details zu vermeiden, aber gleichzeitig präzise genug, um Missverständnisse über die Eigenschaften des betrachteten Objekts auszuschließen.

Sprachen mit grafischer Notationen, wie UML, sind in der Abstraktion stark, während textuelle "Constraint"-Sprachen, wie OCL, die Präzision der Modelle erhöhen. Obwohl diese Sprachpartnerschaften erfolgreich bei der Beschreibung von Systemarchitekturen und -designs eingesetzt wurden, stoßen sie an ihre Grenzen, wenn es um anspruchsvollere Anwendungen geht, die größere Flexibilität und Anpassungsfähigkeit erfordern, ohne Abstraktion und Präzision zu opfern. Solche Anwendungen erfordern Modelle, die über die übliche 2-Level Verteilung von Typ-Objekt hinausgehen. Diese Modelle beherbergen Konzepte die über 3 oder mehr Level instanziiert (Tiefe Charakterisierung) werden können. Das bisherige Forschungsziel bestand darin, das Multi-Level Paradigma in der grafischen Notation umzusetzen, jedoch ist die verbundene Constraintsprache nicht im Fokus der Entwicklung.

Diese Arbeit adressiert dieses Problem, indem sie einen tiefen Dialekt von OCL präsentiert, die mit einer tiefen Modellierungssprache zusammenarbeitet. Dieser Dialekt unterstützt flexible tiefe Modellierungsansätze, ohne die Präzision und Modellstimmigkeit zu beeinträchtigen. Die Arbeit zeigt die Vorteile dieser Technologie anhand von vier Benchmark Challenges zur tiefen Modellierung.

Contents

List of Figures

List of Tables

List of Abbreviations

ASM	Abstract State-Machine
DOCL	Deep Object Constraint Language
DSL	Domain-specific Language
EMF	Eclipse Modeling Framework
EOL	Epsilon Object Language
GLL	Generalized LL
GLR	Generalized LR
GMF	Graphical Modeling Framework
LML	Level-agnostic Modeling Language
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MLM	Multi-Level Modeling
MLT	Multi Level Theory
MOF	Meta-Object Facility
OCA	Orthogonal Classification Architecture
OCL	Object Constraint Language
OMG	Object Management Group
PLM	Pan-Level Model
UML	Unified Modeling Language

Part I

Introduction

This first part of the thesis contains the introduction which outlines the problems addressed by the research, enumerates the requirements that a solution needs to fulfill, defines the hypothesis the thesis explores, and summarises the contributions the thesis makes to the state-of-the-art.

Chapter 1

Motivation

Multi-Level Modeling (MLM) was first proposed over 20 years ago as a way of reducing the accidental complexity involved in modeling domains in which one classification level needs to restrict and characterize concepts in multiple classification levels below, rather than just one level below as in traditional "two-level" modeling approaches [19]. The promise of this approach is evidenced by the large number of MLM languages that have since been developed to support the different MLM approaches and the size of the research community currently working on different aspects of the technology. At the time of writing the number of published MLM approaches is well over 25, with 14 being supported by their own tools [63].

The diversity of the growing collection of MLM languages and the number of active MLM research groups highlights both the strength and weakness of the MLM discipline, however. It is a sign of strength because it shows that MLM addresses a real and growing need inadequately met by traditional "two-level" modeling approaches like the Unified Modeling Language (UML), but it is also a sign of weakness because it shows that the MLM community is struggling to coalesce around a set of common concepts and principles that reduce accidental complexity in the majority of deep characterization scenarios.

A tried and tested strategy for addressing such a language design challenge, where the richness of the proposed language features tends to "hide the wood for the trees", is to separate the modeling features available to users into two mutually supportive and synergistic languages - a core modeling language that focuses on the basic features needed to allow the desired concepts and information to be represented, and an accompanying constraint language that facilitates their application in a rich but controlled variety of ways. The most famous example of such a synergy is the UML and the Object Constraint Language (OCL) [26, 47, 55, 103], in which the latter is a carefully designed constraint language for the former. However, there are many others, such as the UML/Epsilon language family [115], OCL/MOF, OCL/QVT, Schematron/XML [67], or XOCL/XML [102].

To date, most of the research in MLM has focused on core modeling languages, with little attention paid to the accompanying constraint languages. However, to control the large variety of cases and features that evidently needed to be supported in MLM without making the core modeling languages unduly complex, more sophisticated and usable constraint languages are needed that are optimized for the needs of multi-level modeling. Such generally applicable constraint languages are the key to addressing several fundamental problems in current MLM approaches in a concise and straightforward way without making the core language unduly cumbersome.

1.1 Problems

Beyond the use cases of traditional "two-level" object constraint languages, there are three key challenges that multi-level object constraint languages can most effectively address.

P1: Rigid Modeling Styles

The original, motivational use case for MLM was to support the model-based definition and use of domain-specific languages (DSLs) (i.e., to support metamodeling). This use case requires models to adhere to certain rules by which the elements at one level are related to elements at other levels - that is, to be governed by a certain modeling "style". To express a model of a domain of interest (i.e., a domain model) using a language that itself is defined as a metamodel, the former model must be related to the latter model in the following way -

1. instances of elements of the metamodel (which represent components of the DSL's abstract syntax), must exist in the level immediately below,
2. all model elements in a model expressed using the DSL must be an instance of an element in the metamodel at the level above.

However, another common use case for MLM is creating domain models with the primary goal of describing the deep characterization relationships that exist in the domain of interest. In this use case, the above rules are not necessarily appropriate, and styles that allow linguistic extensions [41] and/or leap potency [44] become more natural. The former essentially allows lower levels to include model elements that are not instances of any higher-level element, while the latter allows the instances of a model element to be at any level below that model element, not necessarily the level immediately below.

Evidently, there is a need to provide flexible support for multiple styles of MLM, on top of a common core language, where a particular style defines how the features of the core language should be used and applied in a particular circumstance or to achieve a particular goal.

P2: Ambiguous Modeling Styles

Closely related to the problem of too rigid modeling styles is the problem of ambiguous modeling styles. This is caused by a lack of clarity about what combination of features and rules are appropriate and effective for a particular modeling scenario, rather than by an inability to customize and adapt them. For example, the first of the two relationships identified in the previous subsection is a fundamental pillar of the strictness paradigm which underpins many multi-level modeling approaches. However, the second relationship is not so fundamental. Early MLM papers [19] that advocated the so-called "strictness" doctrine strongly implied that the second relationship was as important as the first, but did not formally require this relationship.

Another example that shows how modeling rules in MLM can evolve and be domain-specific is the deep instantiation mechanism that underpins the deep modeling variant of MLM [17]. This uses the notion of potency to characterize the "type facet" of model elements and control over how many lower levels they can have instances. The first published rules regarding potency were very clear and unequivocal [17] – in every instantiation step (i.e., leading to an "instanceOf" relationship) the potency of the instance has to be one lower than the potency of the type. However, there are certain circumstances when this rule is too rigid – namely when the elements appearing in a generalization set with an abstract subclass are all direct instances of the same model element at the level above. To cope with this special case, Kühne introduced a "relaxed" version of the potency, called "characterization potency" [77] which only requires the potency of a direct instance of a model element to be lower than that elements potency, but not necessarily lower by one. The underlying rule that potency cannot be negative is not changed.

These examples show that the underlying rules governing the use of MLM features in a particular multi-level model not only need to be customizable, but they also need to be clearly and precisely defined in a language that end users can understand.

P3: Unenforced Modeling Patterns

Modeling styles apply to one or more entire levels of a MLM, and often to the whole MLM. However, there are also often smaller groups of model elements that need to be applied in a controlled and uniform way in MLM. While they do not cover all the elements in a level or multiple levels, such so-called "patterns" can potentially involve quite a large number of model elements. A well-known example of an MLM pattern is the powertype pattern which was one of the original motivating factors for MLM. A powertype pattern exists when the instances of one class are subclasses of another class - the former is then referred to as the powertype of the latter. Two main variants are recognized in the MLM literature – the Cardelli variant [27], where the base type is an instance of the powertype, as well as the subclasses of the base type, and the Odell variant [95], where the base type is not an instance of the powertype, only the subclasses of the base type are.

The occurrence of this pattern, in different variants, is often used to illustrate MLM's ability to improve the quality of models by ensuring instances of metamodel elements have the desired properties and relationships. However, the requirement that model elements must adhere to a powertype pattern is rarely explicitly defined and enforced. Moreover, even if it is clear to a modeler that a powertype pattern is appropriate in a particular situation, it is often not clear which particular variant is intended.

The effective use of MLM modelling language therefore requires a clear and enforceable definition of what the use of particular patterns entails. It is possible to address this need by adding dedicated features to core language features to specify pattern applications, such as Multi Level Theory (MLT) [30], but this is then hard-coded into the modeling language and can make it bloated and complex. To solve this, A more flexible and scalable approach is to keep the core MLM language as small as possible and use supporting constraints defined in a suitable MLM-aware constraint language.

1.2 Requirements

To address the aforementioned problems, a deep (i.e., MLM-aware) object constraint language should ideally meet the following requirements.

R1: Precise and Unambiguous

Like any constraint language, a deep constraint language needs to be precise and unambiguous. This means that the language needs to be founded on set theory and first-order logic (predicate calculus). Graphical models alone are usually not precise enough to convey all the information that a well-formed model of a domain needs to convey. A constraint language needs to support and augment the graphical notation to keep models uncluttered and make specifications more precise [120].

R2: Simple and User Friendly

In order to support the basic MLM goal of reducing accidental complexity, a deep constraint language needs to be accessible to mainstream software engineers and domain experts. In other words, it should be easy for them to use, and programming language style syntax should be preferred over mathematical notation.

R3: Conservative Extension of Two-Level Modeling

The language should be a conservative extension to "two-level" modeling languages in the sense that two-level constraints should be a special case of multi-level constraints, and should closely resemble constraints written in established object constraint languages. This means that a constraint on a two-level model written in a two-level constraint language (e.g., OCL) should be directly representable in a deep constraint language with few if any workarounds.

R4: Level Adjuvant

To be multi-level (i.e., deep), the language should not only be "aware" of multiple classification levels, but it should also provide features to enable and simplify constraints supporting deep characterization. This includes features to express the range of levels at which constraints should apply within a stack of classification levels.

R5: Reflective

Finally, as well as supporting the flexible definition of constraints that can enhance the representational richness and subtlety of multi-level models, multi-level constraint languages should also provide linguistic constructs to allow their own features to be queried – that is, they should be reflective. This requires the language definition to be fully modeled within the Orthogonal Classification Architecture (OCA), and the abstract syntax elements to be fully accessible. Thus, making the language aware of the dimensions existing in the OCA. The reflective features of the deep constraint language can access the linguistic dimension by just querying for type name in the linguistic dimension.

1.3 Research Method

The premise underlying this thesis is that it is possible to define and implement a language, as an extension of the OCL, that fulfills requirements R1-R5 for models represented in the core language, in this case, the Level-agnostic Modeling Language (LML). In other words, the research presented in this thesis is based on the following hypothesis –

Hypothesis. "It is feasible to define and implement an OCL-based, deep constraint language to support reflective, level-adjuvant, and dimension-aware constraints on deep (i.e., multi-level) models represented using the LML".

Since the developed language enhances the deep modeling approach supported by the LML, we refer to it as the Deep Object Constraint Language (DOCL).

The research approach used to explore the validity of this hypothesis is the "design science" methodology. To this end, we, therefore, applied the seven design science ingredients of Hevner et al. [60] in the following way –

- *Problem Relevance*: we identified the need for, and the relevance of, the envisaged deep object constraint language.
- *Design as an Artefact*: we constructed a prototype implementation for Melanee, the MLM tool, as a plug-in extension.
- *Design Evaluation*: we evaluated the utility, quality, and efficacy of the developed technology by using it to (a) precisely define, and enforce the use of, well-known multi-level modeling styles and (b) ensure the consistent use of well-known patterns in multi-level modeling.

- *Research Contributions*: we demonstrated the utility of the approach in a range of recognized modeling scenarios (i.e., the various "Challenges" defined in the MULTI workshop series).
- *Research Rigor*: we developed formal models of the syntax of the language and informal descriptions of the semantics of the language's features.
- *Design as a Search Process*: we constructed and evaluated the prototype implementation of the language environment in an agile manner using the established benchmark models in the multi-level modeling community.
- *Communication of Research*: we published key aspects of the language in various international workshops, conferences, and journals.

1.3.1 Foundation Languages

In order to support a multi-level modeling approach that fulfills the aforementioned requirements it is necessary to have (a) a mature multi-level modeling language to serve as the core language upon which the constraint language operates, and (b) a mature, underlying, "two level" constraint language that has already successfully fulfilled the multi-level, agnostic requirements R1, R2, and R3.

Level-Agnostic Modeling Language: The language chosen for (a) is the LML developed at the University of Mannheim because it is one of the richest and most mature MLM languages. Since it is based on the "deep instantiation" mechanism supported by most MLM languages, LML is sometimes characterized as supporting a flavor of MLM known as "deep modeling".

Object Constraint Language: The language chosen for (b) is the OCL, standardized by the Object Management Group (OMG), which is by far the most widely used constraint language in graphical modeling. It was specifically developed to support R1 and R2 and has a mature range of supporting implementations. OCL was also designed to complement the UML as the core language. This matches LML's philosophy of providing features that support UML's "look and feel" wherever possible.

1.3.2 Research Communication

The research conducted as part of this thesis has been presented in various international workshops and journals, including the MULTI workshop at MODELS in 2018, 2019, 2021, 2022, and 2023, the Enterprise Modeling and Information Systems Architectures Journal (EMISAJ), and the International Conference of Systems Modelling and Management (ICSMM).

1.4 Outline

The thesis is split into six parts that group the chapters of this thesis thematically together. The first part, called "Introduction", contains this introduction. The second part, the "Background", includes the foundational topics where key technologies that are the basis of this thesis are introduced and explained. Part three, "DOCL", presents a comprehensive overview of the constraint language developed for the multi-level modeling paradigm. The next part, "Use Cases", presents examples of the application of DOCL in the mentioned modeling environment. In particular, we show what reflective powers the constraint language offers to users and how modeling styles and patterns can be controlled using DOCL. Part five, "Evaluation", contains chapters that deal with solutions to challenges posed by the multi-level modeling community and where DOCL was used extensively to fulfill all requirements. The last part, "Significance", contains the related work and conclusion chapters to contextualize this work in the proper scientific field and to conclude this thesis.

Part II

Background

The second part of the thesis provides an overview of the background technologies upon which the presented solution is founded. The first chapter provides an introduction to the general area of model-driven development, as well as a detailed overview of the OCL which is enhanced in the thesis. The second chapter provides an introduction to the field of multi-level modeling, which is where the thesis makes its contribution, an overview of the Level-agnostic Modeling Language (LML) which the developed technology is designed to complement, and a summary of the Melanee multi-level modeling tool which implements the language developed in the thesis. Finally, the third chapter provides an overview of the theory behind formal languages as well as practical tools to realize them.

Chapter 2

Model-Driven Development

The modern form of model-driven development recognized today emerged in the early 1990s when many of the disparate analysis and design languages existing at the time were merged into the UML [4]. The goal was to raise the level of abstraction at which properties of software systems could be represented, primarily using graphical languages, to increase the productivity of developers [11] and expand the range of stakeholders able to understand the system's features [49]. Model-Driven Development (MDD) is a software development paradigm in which models take center stage in the development process. This contrasts with other development paradigms where models serve primarily as documentation or as intermediate artifacts.

MDD can be described as a two-faceted paradigm [109], that combines the concept of pre-standardised DSLs with transformation engines. The former offers the means to express concepts in customized languages that, in contrast to general-purpose languages, are tailored to the needs of particular domains, while the latter offers the means to transform information modeled in the DSL to instances of other models (i.e., via model-to-model transformations) or to text (i.e., via model-to-text transformations).

The core tenet of MDD is, therefore, that every artifact of the software development process is considered a model [24]. According to Stachowiak [112], models –

1. are representations of natural or artificial originals, which can themselves be models (*mapping*),
2. do not capture all attributes of the original that they represent but are *reductions* that only contain the attributes that are relevant to the creator or user of the model (*reduction*),
3. always have a certain purpose (*pragmatism*).

This definition of model properties is an appropriate foundation for the MDD method.

2.1 The Model-Driven Architecture

MDD covers the general idea of using models to drive the software engineering process. However, there are numerous specific incarnations of MDD that prescribe specific strategies and languages for applying it. One of the most important and well-known is the so-called Model-Driven Architecture (MDA) approach developed and maintained by the OMG – the organization that owns the UML standard.

To cope with the myriad of implementation technologies available today, the MDA emphasizes the separation of the specification of a system's functionality from the details of how it is implemented. This separation of concerns facilitates the creation of platform-independent models (PIMs), which describe the system's functionality in a technology-independent way. The information encoded in them is therefore not tailored to a specific execution platform but instead can be deployed to various execution environments by applying appropriate transformations. More specifically, PIMs can be transformed semi-automatically into different platform-specific models (PSMs) for different execution environments.

The two modeling standards at the core of the MDA are the Meta-Object Facility (MOF) [3] and the UML. The MOF provides a common language and framework for defining and managing models, while the UML provides a graphical notation for creating models that describe software systems and all other kinds of subjects. Platform-independent models are usually represented using the UML, and describe the system at a high level of abstraction, using concepts such as classes, objects, and relationships. The ultimate goal of the MDA approach is to automate the process of creating code from models, using model transformations. Model transformations are rules that define how platform-independent models can be translated into platform-specific models and eventually into code, largely automatically. This approach improves the quality of software artifacts, reduces development costs and time, and enhances maintainability and scalability [74].

2.1.1 The Unified Modeling Language

UML, which lies at the heart of the MDA approach to MDD, was developed with the goal of describing software systems using graphical notations [25]. The current version of UML [4] identifies 14 different types of diagrams,

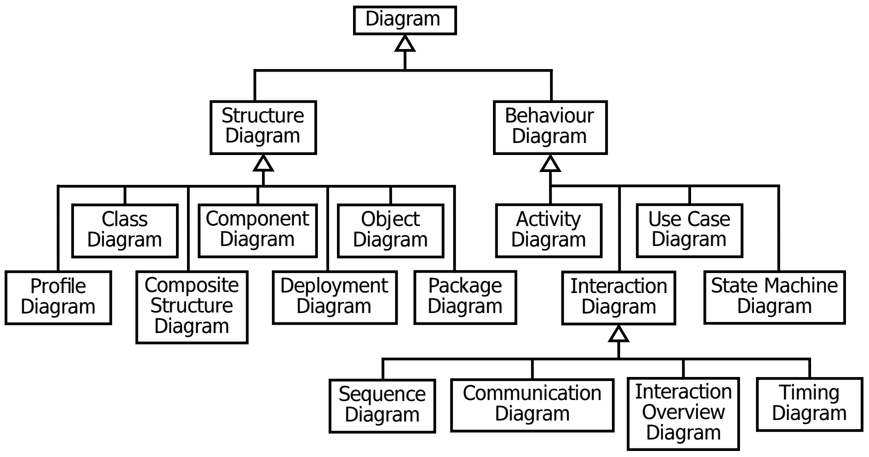


FIGURE 2.1: The 14 UML diagram types

which are called language units. These different diagram types can be categorized into two basic types – structural and behavioral – as shown in Figure 2.1. Behavioral diagram types provide "dynamic" views of systems by describing the step-by-step actions that occur in a software system. A subcategory is interaction diagrams which describe how the flow of information through a software system is controlled. The seven kinds of structure diagrams describe the "static" properties of systems, including their architecture.

2.1.2 The Meta-Object Facility

The MOF is a meta-language for defining domain-specific languages, which since 2005 has been closely aligned with, and integrated into the UML infrastructure. It is the most general meta-model within the MDA and is able to support the definition of any conceivable language. The MOF is also self-describing, i.e., it conforms to itself and serves as the meta-model for the definition of the UML.

Figure 2.2 shows the four-layer meta-model hierarchy of the UML infrastructure [4]. This contains four model levels in which "each (except the top) [is] characterized as an instance of the level above" [11]. The bottom level is the so-called "M0" level and contains the data objects the software is meant to manipulate. The level above is called "M1" and holds the user model, which models the classes of objects (i.e., instances) that can occur at "M0". The next level is called "M2" which is where the UML is defined. Since this level contains a model of the user model at "M1", it is traditionally referred to as a meta-model. The top level is called "M3" and contains the MOF. The MOF is the meta-model for all models at level "M2" [11].

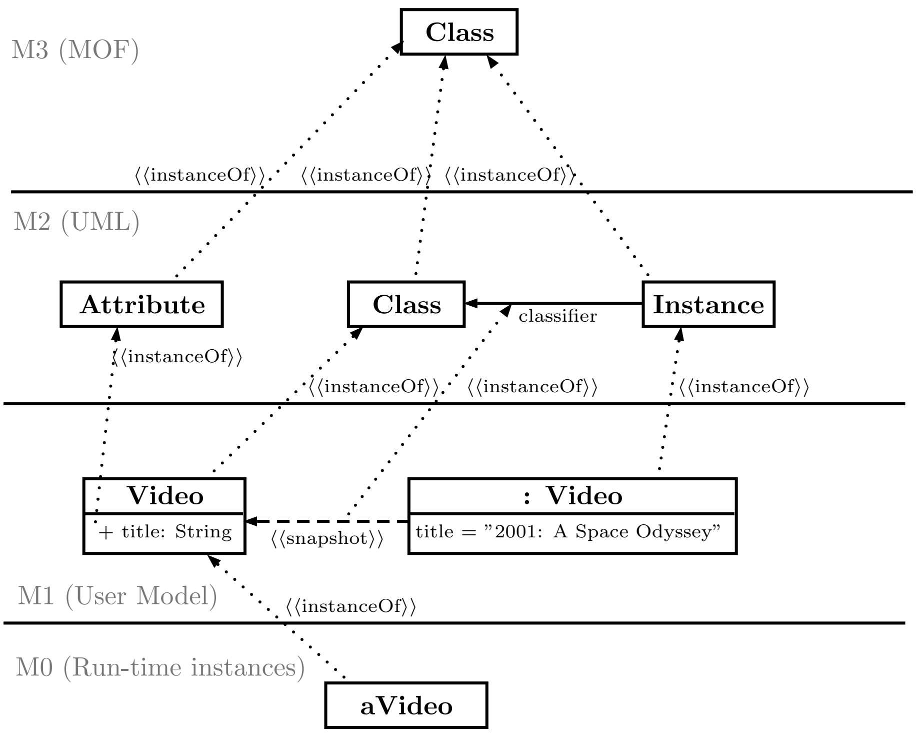


FIGURE 2.2: An example application of the four-layer meta-model hierarchy [4]

2.1.3 Auxiliary MDA Languages

The UML and MOF are augmented by a large suite of additional, auxiliary languages to support the MDA approach. These include but are not limited to the –

Object Constraint Language (OCL): OCL [96] is a textual language used to express constraints and queries on UML models. It provides a way to specify additional rules and conditions that cannot be easily represented through the graphical notations of UML and thus helps enhance the precision and expressiveness of UML models. Since UML version 2, OCL has been an official part of the UML specification and has become the de-facto standard constraint language in MDD.

Query/View/Transformation (QVT): QVT [3] is a model transformation language. It defines a set of transformation operations to manipulate and convert models from one representation to another. QVT enables the automatic generation of code or other artifacts from models. These models must conform to the MOF so that the MOF can support three categories of transforming operations (or use cases) in a modeling environment – transformation, query, and view operations.

XML Metadata Interchange (XMI) Standard: XMI [122] is a standard for exchanging metadata information in XML format. It provides a way to serialize models represented in UML or other MOF-compliant languages and transfer them between modeling tools or repositories.

Common Warehouse Metamodel (CWM): CWM [101] is a language for modeling and exchanging metadata about data warehouses and business intelligence systems. It provides a standardized way to represent and manipulate metadata related to data integration, data transformation, and data mining.

UML Profile Mechanims: UML profiles [7] allow the UML to be tailored to particular specific domains, industries, or platforms. By using the profiling mechanism, it is possible to define sets of UML elements, relationships, and stereotypes tailored to the modeling needs of a particular problem domain or application. UML profiles provide a way to simulate the extension of the UML's metamodel and notation to accommodate domain-specific concepts and modeling constructs. Key examples of standard profiles include SysML [58] (for systems engineering), BPMN [32] (for business process modeling), MARTE [116] (for modeling and analysis of real-time and embedded systems), CCM [88] (for component-based programming), EDOC [51] (for building PIMs of enterprise applications), EAI [51] (for application integration and building loosely-coupled systems), QoS [40] (for modeling quality of service and fault tolerance requirements), and UTP [22] (for supporting automated testing in MDA-based development environments).

2.2 Object Constraint Language

The focus of this thesis is the OCL, which is designed to be used in tandem with an accompanying modeling language like the UML rather than in a stand-alone setting. Every expression in the OCL is therefore made in the context of a model element expressed in the accompanying modeling language, usually a class specification, and constrains the values that can be assigned to that model element, or related model elements, in instances of the model [120]. OCL expressions are not allowed to alter the underlying model in any way [120] and are thus "declarative". The language is used to make models more precise [110] and add information that would otherwise be overwhelming when expressed in a visual notation.

Unfortunately, when reasoning about UML/OCL models, OCL expressions are generally undecidable [23]. This means that the termination of reasoning algorithms or the delivery of correct output can not be ensured. The root of the problem is that the expressiveness of OCL goes beyond first-order logic [97]. However, under certain circumstances, OCL expressions can be made decidable (see Section 15.1.6).

Models, in general, are an abstraction of the problem domain and, therefore, can be incomplete, informal, and imprecise [119]. This has the advantage they are simple and easily understandable for most people, especially when portrayed in graphical notations. However, it is also a reason why an additional language (often textual) is needed to add precision to the models. The combination of the graphical (easily understandable) and the textual (more complex) languages offers the best of both worlds to all the stakeholders in a model [119].

OCL's approach for specifying behavior is based on the "Design by Contract" principle [91] which, in turn, is inspired by the legal notion of a contract. According to this principle, software components should define their expected behavior in the form of contracts that their operations (a.k.a. methods) enter into when invoked. An operation's contract specifies the preconditions, postconditions, and invariants that must be satisfied for it to be judged as behaving correctly. Preconditions specify what must be true before a particular operation can be invoked, while postconditions specify what must be true after the operation has been performed (assuming the precondition was true). Invariants are conditions that must hold true at all times, whenever an operation is not executing.

By using contracts to specify the behavior of software components, the OCL helps to ensure that software is reliable, robust, and easy to maintain. In addition, because contracts are expressed in a formal language, they can be used to automatically generate test cases and verify the correctness of software implementations.

The root of every OCL expression is the context (usually a class) it is defined in. From the context, users can "navigate" over the model by, for example, accessing attributes or traversing connections. When the user navigates the model from the context to another connected class the result is a collection. Depending on the multiplicities of the navigated connections, the result is either a *Set* or a *Bag*. A *Set* is a collection in which every member is unique (i.e., has no duplicates), whereas a *Bag* is a collection that allows duplicates. Given the nature of associations in the UML, the first navigation in a navigation chain always results in a *Set*. Further, navigation, in general, results in a *Bag*. Since a set is a specific case of a bag, it is important to be clear whether a resulting collection is truly a *Set* or a *Bag*. In total, there are four collection types in the OCL – *Set*, *Bag*, *OrderedSet*, and *Sequence*.

The basic building blocks of OCL expressions are objects and object features. Each object has a particular type from which it was instantiated and therefore offers the corresponding set of features. OCL recognizes two categories of types, user-defined model types and predefined types such as *Integer*, *Real*, *String*, and *Boolean*, as well as the aforementioned collection types. User-defined types are the classes in the accompanying model. Every class or interface the user introduces in the model is automatically an OCL type that can be referenced in OCL expressions so that its attributes, operations, and associations can be used for navigation.

As well as being either user-defined or predefined, OCL types are also either object types or value types. An object type is a type that represents a class of objects, while a value type is a type whose "instances" represent specific values. In OCL, object types can be used to specify constraints on the behavior of their instances, such as preconditions and postconditions of operations or invariants that must be satisfied at all times. Object types can also specify constraints on relationships between objects, such as associations and aggregations.

A value type, on the other hand, is a type whose "instances" represent specific values, such as a number or a specific string. Value types do not have state or behavior, and they are typically used to specify constraints on the values of attributes or parameters. Object types and value types are represented using the same syntax. However, their semantics are fundamentally different, and it is important to understand the distinction when specifying constraints in OCL.

As mentioned previously, the "Design by Contract" principle is the inspiration for three kinds of constraints available to modelers in OCL, invariants, pre- and postconditions. Overall, OCL supports seven distinct kinds of constraints -

- Invariant constraints
- Pre constraints
- Post constraints
- Body constraints
- Definition constraints
- Init constraints

- Derive constraints

A *Body* constraint has an operation as its context and defines the behavior of that operation in terms of input parameters and return values. These can either be individual OCL types or sets of types. A *Definition* constraint allows new attributes and operations to be introduced into a model without using the associated graphical language. The context of such a constraint is a class or an interface. When adding a new attribute, a definition constraint can also assign a value to it via an expression of the appropriate type. A new operation can be added in basically the same way but the return type is not restricted to OCL's predefined types.

An *Init* constraint allows a modeler to assign initial values to an attribute. The context is always the attribute that is assigned that initial value, which is specified in the associated expression. Instead of assigning a value to an attribute, values can also be derived and changed depending on the state of the whole system. The *Derive* constraint is also defined in the context of an attribute and derives the value of an attribute depending on an expression.

Every model type in OCL is regarded as a subtype of *OclAny* and an instance of *OclType*. *OclAny*, defines a range of operations which are therefore available on all object types in a model –

- object1 = object2 (equals)
- object1 <> object2 (unequals)
- object.oclIsUndefined()
- object.oclIsKindOf()
- object.oclIsTypeOf()
- object.oclIsNew()
- object.oclAsType(object2)
- object.oclInState(state)
- object.allInstance()

Every operation in this list, with the exception of *oclAsType()* and *allInstances()*, has the return type "Boolean". The *oclAsType()* operation returns a cast of the object to another type (i.e., the return type is the type that was passed in the parameter) as long as the object is castable into the desired type.

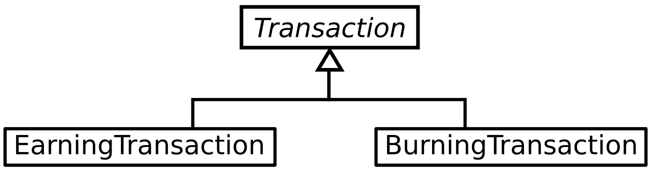


FIGURE 2.3: The Earning/Burning Transaction snippet of the Royal & Loyal example [120]

The *allInstances()* operation returns a set of instances of the class concerned. This operation has to be handled with care because it is usually impossible to estimate how big the resulting set is going to be(i.e., how many instances will exist when this operation is executed). If possible this operation should therefore be avoided.

The *oclIsUndefined()* operation checks if the object in question is defined(i.e., not null or of type *oclVoid*). This type is returned to indicate when an operation fails due to incorrect type casting or when an attempt is made to get an element from an empty collection.

Finally, the *oclIsTypeOf()* and the *oclIsKindOf()* operations are used to check what classes in an inheritance hierarchy an object is a direct or indirect instance of. The *oclIsTypeOf()* operation takes a class as an input parameter and returns true if the instance is a direct instance of this type. The *oclIsKindOf()* operation is the same, except it returns true if the instance is a direct or indirect instance of the parameter.

```
context BurningTransaction
    self.oclIsKindOf(Transaction) = true
    self.oclIsTypeOf(Transaction) = false
    self.oclIsKindOf(BurningTransaction) = true
    self.oclIsTypeOf(BurningTransaction) = true
    self.oclIsKindOf(EarningTransaction) = false
    self.oclIsTypeOf(EarningTransaction) = false
```

CONSTRAINT 2.1: Truth values for the oclIsKindOf() and oclIsTypeOf operations in the context of BurningTransaction

This difference is illustrated by the six different expressions shown in Constraint 2.1 based on the small class diagram shown in Figure 2.3. All six expressions are evaluated in the context of *BurningTransaction*, and check whether the type passed as a parameter is a direct (*OclIsTypeOf*) or indirect (i.e., *OclIsKindOf*) type of an instance of *BurningTransaction*. A direct type of an instance is also regarded as being an indirect type of that instance, but not vice versa.

The first expression returns true because *Transaction* is a direct type of *BurningTransaction*, while the second is false because *Transaction* is not a direct type of *BurningTransaction*. When the parameter is *BurningTransaction* both *OclIsTypeOf* and *OclIsKindOf* return true because *BurningTransaction* is both a direct and an indirect type of *BurningTransaction*. On the other hand, when the parameter is *EarningTransaction* both *OclIsTypeOf* and *OclIsKindOf* return false because *EarningTransaction* is neither a direct nor an indirect type of *BurningTransaction*.

2.2.1 Collection Operations

As mentioned previously, in OCL there are four different collection types: Set, Bag, Sequence, and OrderedSet. Each has a set of predefined standard operations that can be used to manipulate and iterate over the elements in the collection.

The standard collection operations in OCL are categorized into five groups based on the type of operation they perform –

- *Basic operations*: These operations include "includes", "excludes", "union", "intersection", and "difference" and allow elements to be added or removed from a collection, or two collections to be combined into a single collection.
- *Comparison operations*: These operations include "equals" and "notEquals" and allow two collections to be compared to see if they contain the same elements.
- *Set operations*: These operations include "isSubsetOf", "isProperSubsetOf", "symmetricDifference", and "flatten" and allow set operations to be performed on collections, such as determining whether one collection is a subset of another or computing the symmetric difference between two collections.
- *Ordering operations*: These operations include "first", "last", "indexOf", "at", and "subSequence" and allow elements to be retrieved from a collection based on their position or index.
- Iteration operations: These operations include "forAll", "exists", "select", "reject", "collect", and "iterate" and make it possible to iterate over the elements in a collection and perform various operations on them, such as filtering them based on a condition or transforming them into a new collection. The *iterate* operation is a special case because it can be used to define the semantics of every other iteration operation. It is the

most general iteration operation that is used for complex constraints that are not expressible using the other standard iteration operations.

These are just a few examples of the standard collection operations available in OCL. By using these operations, it is possible to manipulate and query collections in powerful and flexible ways, and to work with complex data structures in OCL models.

All of the OCL features and operations presented in this chapter were defined for, and only work in, a two-level setting. That means that constraints only apply to the instances of the model elements that form the context of the constraint - i.e., at the level immediately below. OCL constraints on a UML class diagram are evaluated on instances of the classes specified in the class diagram.

Chapter 3

Multi-Level Modeling

The MLM paradigm was developed to address the limitations caused by the traditional assumption that models exist in the context of only two levels – the “model” level, containing model elements that define the classes of objects that can exist, and the “instance” level, where those objects reside. The objects at the instance level are thus “instances of” model elements at the model level. Many aspects of the UML notation, such as the class naming notation, the dichotomy between attributes and slots, the dichotomy between associations and links, etc., make it difficult and awkward to represent models existing within a multi-level model stack (i.e., that are both models and instances at the same time) [19].

Multi-level modeling languages were developed to address this problem by offering level-agnostic modeling syntax (both abstract and concrete) that recognizes the duality of many of the modeling concepts comprising (intermediate levels of) multi-level models and by providing features that allow types to characterize more than just their immediate instances [19].

Since the emergence of the MLM paradigm, many different language dialects and variants have emerged that occupy different niches in the MLM community and specialize in different application fields. The thesis focuses on an approach often characterized as “deep” multi-level modeling or “deep modeling” for short.

3.1 Deep Modeling

As several authors have pointed out, traditional attempts to scale up classic two-level modeling languages to accommodate multiple model levels have fundamental problems. The most famous attempt is the venerable four-layer “UML infrastructure” defined by the OMG to support the Model-Driven Architecture (MDA) built around a "meta-modeling” approach to language design and extension [19, 41].

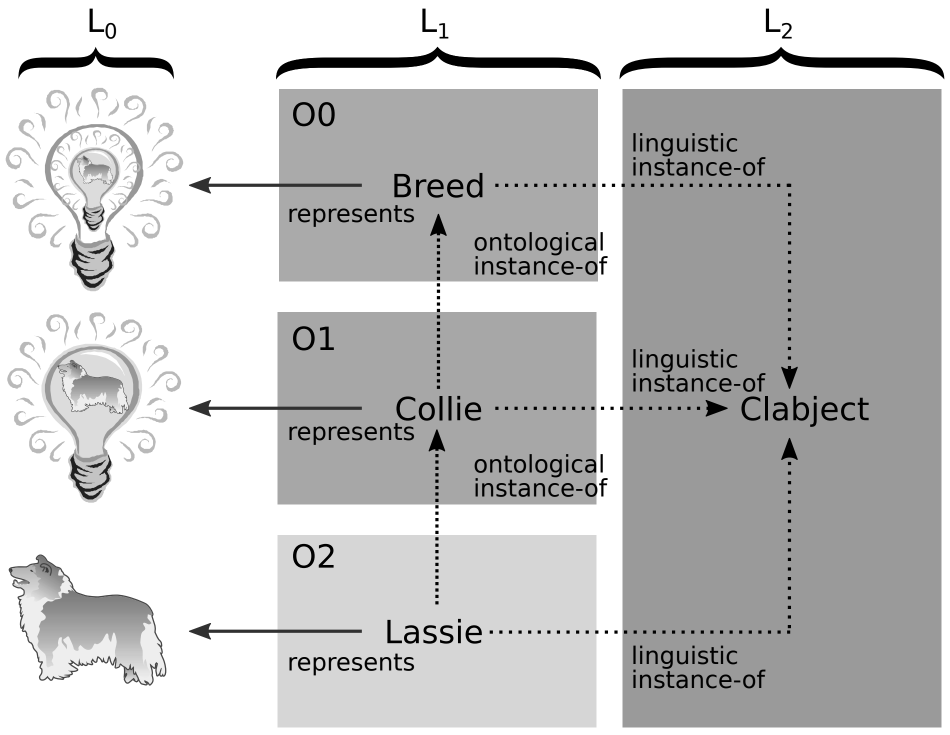


FIGURE 3.1: Orthogonal Classification Architecture (OCA) [11]

When a language like the UML, fundamentally designed to support only two classification levels, is used to model domains that naturally span more than two levels, modelers are often forced to compress the representation of multiple domain levels into just two model levels. This invariably requires the use of so-called "workarounds" that make the resulting models overly complex [41] and increase the accidental complexity they contain [19]. Deep modeling was designed to address this problem.

3.1.1 Orthogonal Classification Architecture

One of the fundamental principles of deep modeling is to recognize two fundamentally different kinds of classification relationships and organize their content along two orthogonal dimensions. This gives rise to the so-called "Orthogonal Classification Architecture" (OCA) illustrated schematically in Figure 3.1. The dominant dimension is the linguistic dimension, shown horizontally in Figure 3.1, which captures how (i.e., in what form) concepts in the domain of interest are modeled (e.g., classes/objects, attributes/slots, association/links, etc.). The other dimension is the ontological dimension, shown vertically in Figure 3.1, within which a single level of the linguistic dimension (i.e., L_1) represents multiple, domain classification levels. The linguistic meta-model is depicted on the right-hand side of Figure 3.1. Every model element in L_1 is a linguistic instance of *Clabject*.

In this example, the model has three ontological levels called O_0, O_1, and O_2. The first ontological level, *O0*, shown at the top contains the *Breed* entity, a linguistic instance of *Clabject*. The second level accommodates the *Collie* entity, which has two types, an ontological type, *Breed*, and a linguistic type, *Clabject*. The third level contains the *Lassie* entity which also has two types, an ontological type, *Collie*, and a linguistic type, *Clabject*. The model elements in L_1 represent the concepts and objects in the real world, which are stored in L_0. The relationships between L_1 and L_0 are thus "represents" relationships rather than a classification relationship, just like the relationship between the bottom two levels of the MDA infrastructure.

When expressed using a language that supports deep instantiation, models are more capable of capturing the complexity of domains, representing diverse relationships, and accommodating hierarchies that extend beyond a single level. This flexibility enhances the utility and power of models in many application domains, such as knowledge representation, reasoning, and semantic querying.

3.1.2 Deep Classification

Deep characterization refers to the description of relationships and constraints between properties, attributes, and/or features of elements across multiple classification levels [81]. In other words, it involves the capturing and representing of domain characteristics that extend beyond a single level of abstraction or classification [19]. In multi-level modeling, deep characterization facilitates a more nuanced and comprehensive representation of model elements by considering their attributes or properties in relation to higher-level concepts and their associated instances. It goes beyond surface-level descriptions and enables a richer understanding of the relationships and distinctions between elements at different levels of classification and abstraction. Deep characterization is therefore usually employed in situations where a complex system or domain requires finer-grained distinctions and a more detailed representation of the entities involved. Among other things, it allows specific characteristics that are relevant and applicable across multiple levels of classification [81] to be represented concisely and precisely.

3.1.3 Deep Instantiation

Deep instantiation, as opposed to "shallow instantiation", is one mechanism for supporting deep characterization used by numerous multi-level modeling languages. It uses an additional concept, such as potency, to describe the ability of a class to govern the form of its offspring over multiple classification levels [19]. Shallow instantiation, the classic two-level modeling approach, cannot accommodate classification hierarchies that are deeper than one level.

With shallow instantiation, there is a clear separation between classes (i.e., abstract concepts in a model) and their instances. Instances are directly related to classes, but there is no provision for creating instances of instances of classes. This limitation severely limits model expressiveness when representing classification hierarchies that naturally have multiple levels or when there is a need to represent more complex relationships between instances. Deep instantiation, on the other hand, enables the creation of instances within a multi-level classification hierarchy, providing greater expressiveness and flexibility in representing the relationships and structures naturally occurring in many real-world domains.

3.2 The Level-agnostic Modeling Language

The Level-agnostic Modeling Language (LML), developed by the Software Engineering Group at the University of Mannheim [14], is an OCA-based deep modeling language whose semantics and syntax (both abstract and concrete) were designed to be –

1. fully level-agnostic, so that there are no notational or representational differences in the way a model element is represented just because of the level it occupies. In other words, all concepts are represented in the same way regardless of what level they occupy.
2. as UML-like as possible, consistent with the previous goal, so that LML models have a look-and-feel that experienced UML modelers will be as familiar with as possible.

Since it is defined in the context of the OCA, LML's metamodel corresponds to *L2* of the linguistic dimension. In [72], where it was originally defined, this is called the Pan-Level Model (PLM) since it defines the concepts that span the ontological levels. The PLM, shown in Figure 3.2, is thus

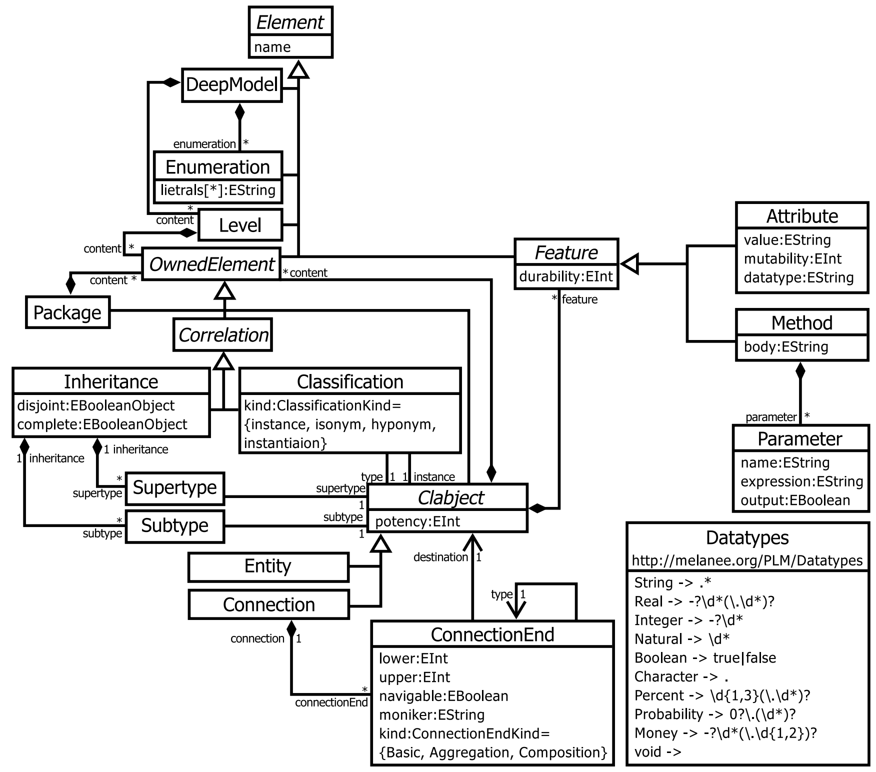


FIGURE 3.2: The Pan-Level Model (PLM) [72]

the metamodel for LML and defines the abstract syntax available for representing deep models. To give an example of LML's use and provide a setting to explain its main features, Figure 3.3, 3.4, 3.5 and 3.6 present an LML "solution" to the so-called "Process Challenge" that was published in the EMISAJ special issue [83].

Element: The most abstract construct in the PLM level is the class *Element*, from which every class inherits directly or indirectly, except *Subtype*, *Supertype* and *Package*).

DeepModel: This class serves as the container for every model element the user can create in a multi-level model. It directly contains the classes *Level* and *Enumeration*. In a *DeepModel*, levels are connected via the *content* composition relationship. An example of a *DeepModel* is the collection of diagrams in Figure 3.3 through 3.6.

Level: The *Level* concept is used to create ontological levels in a *DeepModel* and can be instantiated an arbitrary number of times. It serves as the container for *Clabjects*, *Correlations*, and *Features*. These can be accessed from a

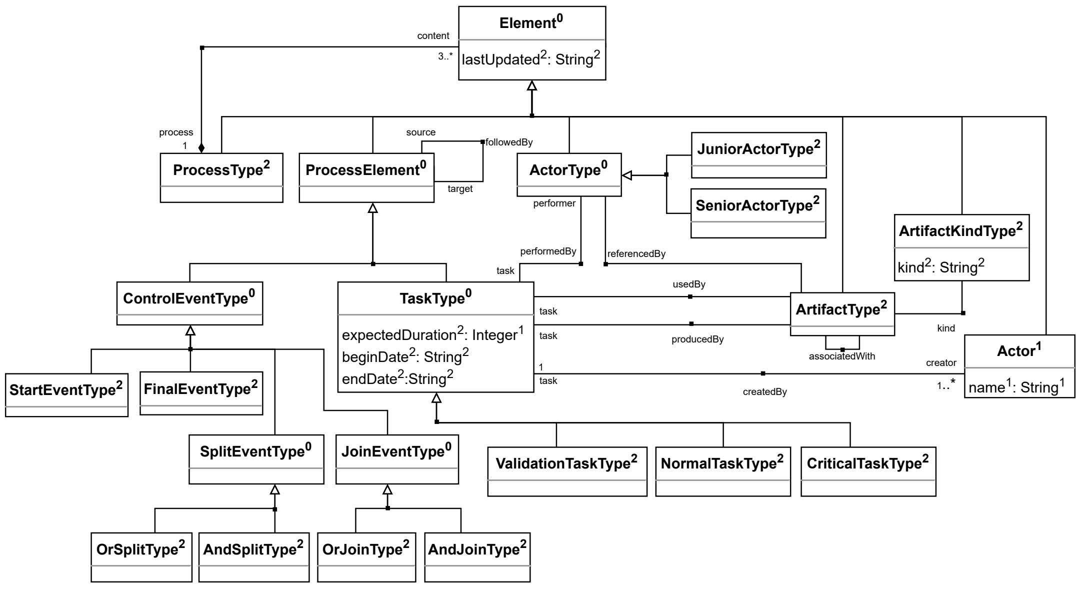


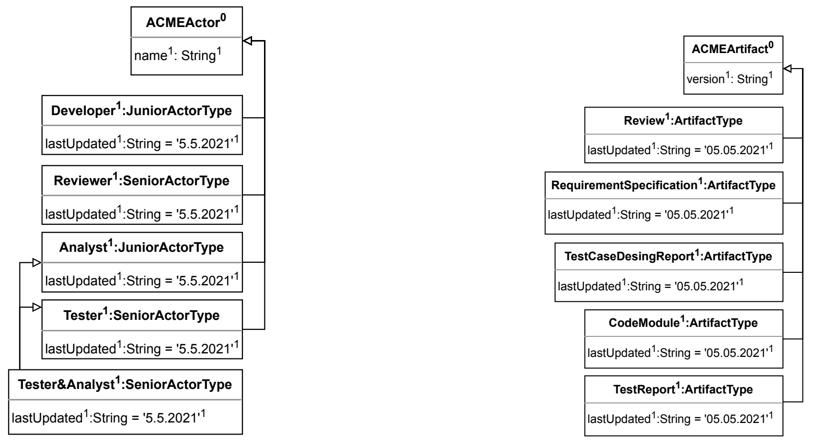
FIGURE 3.3: Level O_0 of the Process Challenge

level via the *content* relationship. For example, Figure 3.3 corresponds to one level in the aforementioned deep model.

Clabject: The *Clabject* is the core concept of multi-level modeling. The name is a composition of "Class" and "Object" ("Cla-bject"), to highlight the fact that it is meant to capture the duality of these concepts. A clabject is both a class (i.e., has a class facet) and an object (i.e., has an object facet) at the same time. A clabject can serve as a type for other clabjects and be an instance of another clabject. In the meta-model, the class *Clabject* contains *Features* which are specialized into *Attributes* and *Methods*.

Entity: This class is one of the two concrete sub-classes of *Clabject*. This concept is related to the *Class* and *Object* concepts in UML since it represents both types and individuals. In Figure 3.3 the model element called *ProcessType* is an instance of *Element*.

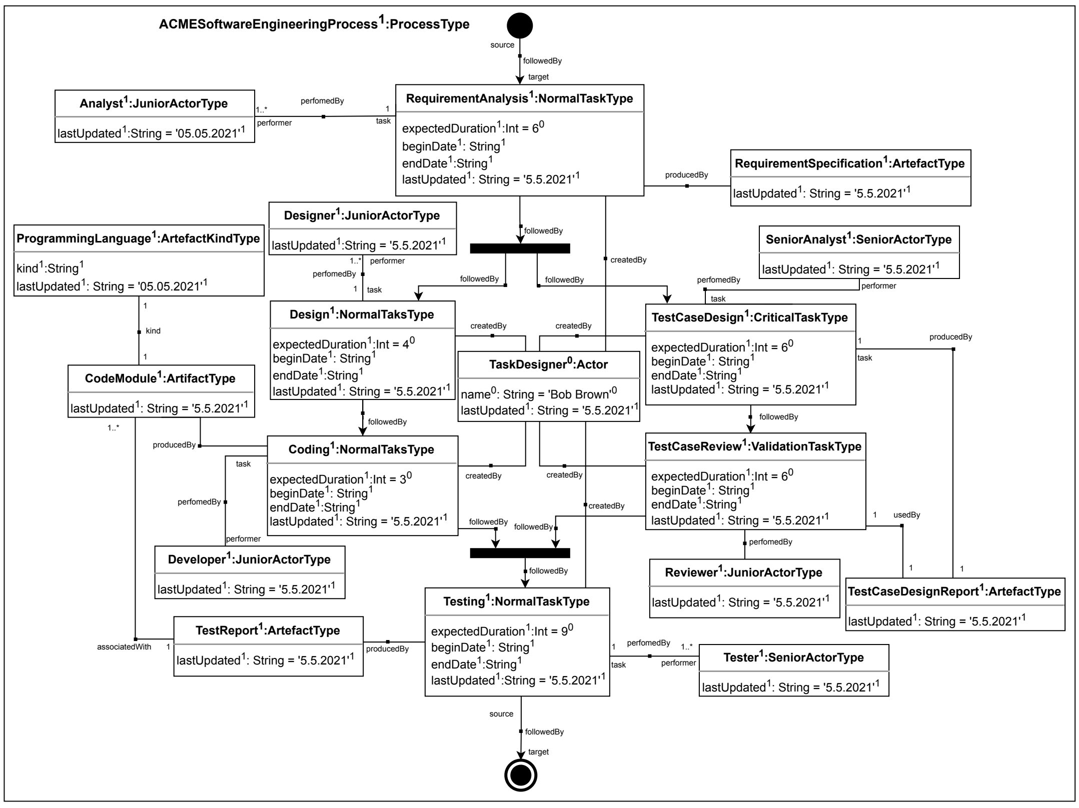
Connection: The *Connection* concept is the other concrete sub-class of *Clabject*. It corresponds to association classes in UML and contains attributes or methods. Connections are connected to entities via *ConnectionEnds*. At least two connection ends must exist within a connection instance. Connection ends can also have different semantics. There are "normal" connection ends that connect clabjects with each other without special semantics. The connection between *TaskType* and *ActorType* in Figure 3.3 is an example. A filled black diamond at the end of a connection represents a "composition".



ACMEActor0
name1: String1
Developer1:JuniorActorType
lastUpdated1:String = '5.5.2021'1
Reviewer1:SeniorActorType
lastUpdated1:String = '5.5.2021'1
Analyst1:JuniorActorType
lastUpdated1:String = '5.5.2021'1
Tester1:SeniorActorType
lastUpdated1:String = '5.5.2021'1
Tester&Analyst1:SeniorActorType
lastUpdated1:String = '5.5.2021'1
ACMEArtifact0
version1: String1
Review1:ArtifactType
lastUpdated1:String = '05.05.2021'1
RequirementSpecification1:ArtifactType
lastUpdated1:String = '05.05.2021'1
TestCaseDesingReport1:ArtifactType
lastUpdated1:String = '05.05.2021'1
CodeModule1:ArtifactType
lastUpdated1:String = '05.05.2021'1
TestReport1:ArtifactType
lastUpdated1:String = '05.05.2021'1

(A) Actor powertype (B) Artifact powertype

FIGURE 3.4: Powertype definition in Level O_1 of the Procecss Challenge



ACMESoftwareEngineeringProcess1:ProcessType
source
followedBy
target
Analyst1:JuniorActorType
lastUpdated1:String = '05.05.2021'1
perfomedBy
performer
task
RequirementAnalysis1:NormalTaskType
expectedDuration1:Int = 60
beginDate1: String1
endDate1:String1
lastUpdated1: String = '5.5.2021'1
producedBy
RequirementSpecification1:ArtefactType
lastUpdated1: String = '5.5.2021'1
Designer1:JuniorActorType
lastUpdated1:String = '5.5.2021'1
ProgrammingLanguage1:ArtefactKindType
kind1:String1
lastUpdated1: String = '05.05.2021'1
SeniorAnalyst1:SeniorActorType
lastUpdated1: String = '5.5.2021'1
createdBy
kind
Design1:NormalTaksType
expectedDuration1:Int = 40
beginDate1: String1
endDate1:String1
lastUpdated1: String = '5.5.2021'1
TaskDesigner0:Actor
name0: String = 'Bob Brown'0
lastUpdated1: String = '5.5.2021'1
TestCaseDesign1:CriticalTaskType
expectedDuration1:Int = 60
beginDate1: String1
endDate1:String1
lastUpdated1: String = '5.5.2021'1
CodeModule1:ArtifactType
lastUpdated1: String = '5.5.2021'1
Coding1:NormalTaksType
expectedDuration1:Int = 30
beginDate1: String1
endDate1:String1
lastUpdated1: String = '5.5.2021'1
TestCaseReview1:ValidationTaskType
expectedDuration1:Int = 60
beginDate1: String1
endDate1:String1
lastUpdated1: String = '5.5.2021'1
usedBy
Developer1:JuniorActorType
lastUpdated1: String = '5.5.2021'1
Reviewer1:JuniorActorType
lastUpdated1: String = '5.5.2021'1
TestCaseDesignReport1:ArtefactType
lastUpdated1: String = '5.5.2021'1
associatedWith
TestReport1:ArtefactType
lastUpdated1: String = '5.5.2021'1
Testing1:NormalTaskType
expectedDuration1:Int = 90
beginDate1: String1
endDate1:String1
lastUpdated1:String = '5.5.2021'1
Tester1:SeniorActorType
lastUpdated1: String = '5.5.2021'1

FIGURE 3.5: Level O_1 of the Process Challenge

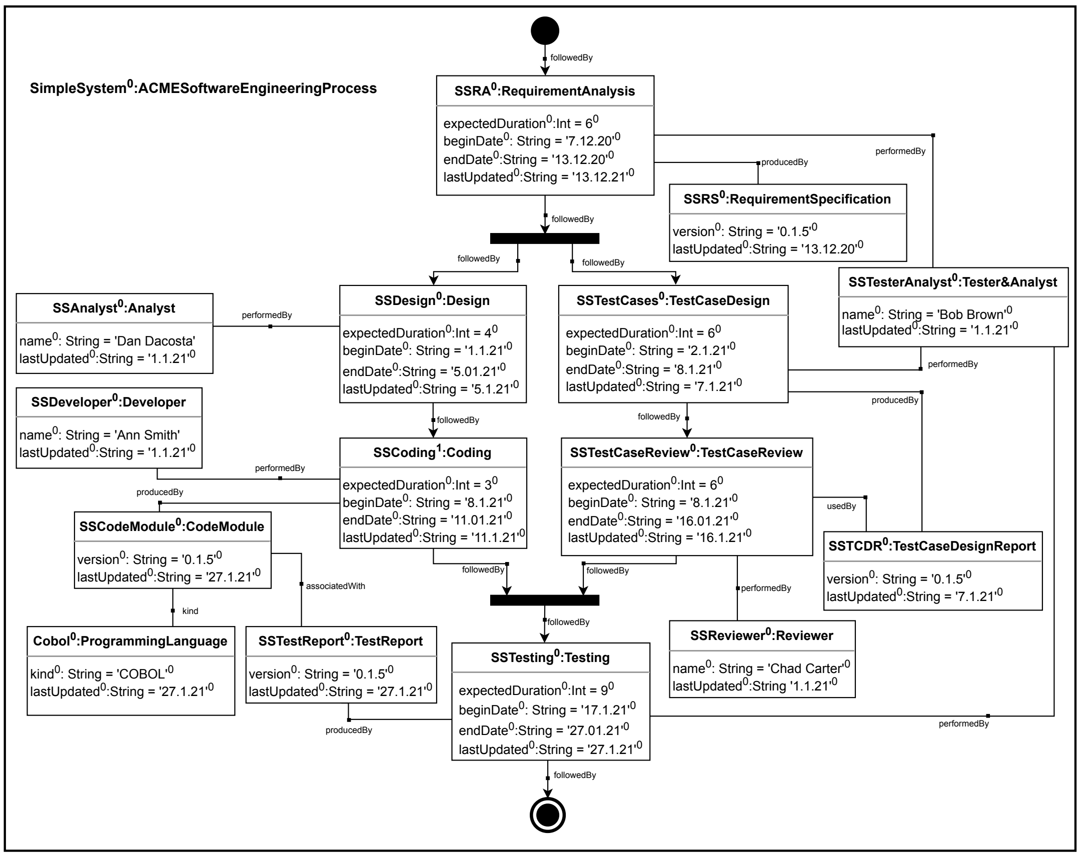

SimpleSystem⁰:ACMESoftwareEngineeringProcess
SSRA⁰:RequirementAnalysis
SSRS⁰:RequirementSpecification
SSAnalyst⁰:Analyst
SSDesign⁰:Design
SSTestCases⁰:TestCaseDesign
SSTesterAnalyst⁰:Tester&Analyst
SSDeveloper⁰:Developer
SSCoding¹:Coding
SSTestCaseReview⁰:TestCaseReview
SSCodeModule⁰:CodeModule
SSTCDR⁰:TestCaseDesignReport
Cobol⁰:ProgrammingLanguage
SSTestReport⁰:TestReport
SSTesting⁰:Testing
SSReviewer⁰:Reviewer
followedBy
performedBy
producedBy
associatedWith
kind
usedBy

FIGURE 3.6: Level O_2 of the Process Challenge

It implies a multiplicity value of 1 on that connection end. The connection between *Element* and *ProcessType* shows this type of connection. A white diamond represents "aggregation" semantics.

ConnectionEnd: This class represents the endpoint of connection navigation. As well as being the link between *Entity* and *Connection* classes, it holds the multiplicity information and also whether the connection is navigable and has a role name. Additionally, there are three types of connection ends, e.g., basic, composition, and aggregation. In Figure 3.3, the connection between *Element* and *ProcessType* has two endpoints. Each of the endpoints is annotated with a moniker and the multiplicity values. The connection end that points to *ProcessType* uses *process* as its moniker and a multiplicity value of 1. The other connection end has a moniker called *content* and the multiplicity values are '3' and '*'.

Attribute: This concept is a specialization of *Feature* and represents properties of a *Clabject*. Attributes are the key enabler of the "deep instantiation" mechanism, which means that an attribute can be part of a clabject's intension over more than one instantiation step. Attributes have a name, a type, and a value. A value can be assigned to attributes that can still be "instantiated" (attributes do not participate in classification relationships, but are regarded as being part of a clabject's intension). The clabject *TaskType* in Figure 3.3 contains three attributes that are called *expectedDuration*, *beginDate*, and *endDate*.

The UML uses the term "attribute" to refer to property types and the term "slot" to refer to property values. To unify them, the original paper that introduced potency suggested the term "field" to cover both attributes and slots and the term "dual field" to refer to "deep attributes" (i.e. non-slot fields) that have a value as well as a type [19]. However, simple fields (attributes without values) and dual fields (attributes with values) were regarded as existing in separate instantiation chains. This means an instance of a simple field could not be a dual field – a capability used quite frequently in practice [84]. For simplicity, therefore, LML removes this distinction and regards all attributes and slots (i.e. all fields) as having the same general form (i.e. a name, a type, and a value). However, the type and the value need not always be shown[1]. The LML refers to this unified concept as a "deep attribute", or simply "attribute" for short. In the rest of this work, we will use the term "attribute" to mean "deep attribute".

Method: The *Method* class is the other specialization of *Feature*. A method consists of a method body and parameters. *Methods* are similar to *Attributes* and are also contained by clabjects.

Classification: The only kind of relationship that can cross levels is the *Classification* relationship. This is a 1-to-1 relationship connecting a clabject representing an ontological instance to a clabject representing its direct ontological type. Since no other concept in the meta-model is allowed to cross a level boundary it is therefore essential to multi-level modeling. Classification relationships can be represented graphically, as a dashed line, and textually following the name of a clabject using the traditional UML ":" notation. For example, in the *O1* level of the process model (Figure 3.4), the *Developer* clabject is declared as being of type *JuniorActorType* and therefore in a classification relationship with it. The colon separates the name and the type of the clabject.

[1]Just as the type of an attribute need not always be shown in the UML

Inheritance: LML supports generalization sets through the *Inheritance* concept. As in UML, inheritance can be disjoint or overlapping, or complete or incomplete. An inheritance model element contains a pointer to the super- and sub-types participating in a generalization, which are *Clabject* instances. *Connections* can also participate in inheritance relationships, therefore. Inheritance declares a generalization/specialization relationship between clabjects, which means that all the instances of a subclass (or specialization) clabjects are also instances of the superclass (or generalization) clabjects.

In LML, inheritance relationships can only exist between elements at the same ontological level. The basic semantics of specialization is captured by the notion of substitutability, which is sometimes said to be motivated by the Liskov Substitution principle [87]. This basically states that wherever an instance of one class is expected, an instance of a subclass can be provided in its place as long as it still fulfills the contract with the user. In terms of set theory, this means that all possible instances of a subclass must satisfy the intension of the superclass. so that the extension of the subclass is a subset of the extension of the superclass [28]. This implies the intension of the subclass must subsume the intension of the superclass in all possible worlds. It is not sufficient for the intension of the subclass to allow some instances that satisfy the intension of the superclass to be created, it must exclude all instances that do not satisfy it.

This definition means that there is a natural distinction between the “direct” type of a clabject and the “indirect types” of a clabject. For instance, the clabject *Developer* in Figure 3.4a is a direct instance of *JuniorActorType* and an indirect instance of *ActorType* and *Element* in Figure 3.3. The *ActorType* and *Element* clabjects are indirect types for the *Devloper* clabject and both types are abstract clabjects, so they can only have indirect instances and no direct instances because they are impotent (i.e., have potency 0 values).

The distinction between direct and indirect instances should not be confused with the distinction between instances and “offspring” of a clabject. The offspring of a clabject are instances of the instances. For example, the clabject *SSAnalyst* in Figure 3.6 is an instance of *Analyst* of the level directly above but offspring of *JuniorActorType*. The offspring of a clabject can be both direct and indirect.

In Figure 3.3 an example of an inheritance relationship is shown between *Element* and *ProcessType*. In fact, all other clabjects are subclasses of *Element*. In this level, the *Element* clabject is the root.

3.2.1 Vitality Properties

In LML, the deep instantiation mechanism that supports deep characterization is governed by three so-called vitality properties. Two of these, potency and mutability, are used in other multi-level modeling languages, although sometimes in different forms. However, the combination of the three is unique to LML.

Potency: Potency is a non-negative integer used to specify the degree to which an entity is a type and/or an object in a level-agnostic way. It basically indicates over how many levels a clabject can be instantiated and influence direct instances [41]. Early rules for potency stated that when a clabject is instantiated, the potency of the instance has to be exactly 1 less than the potency of the type [19]. These rules were developed with constructive modeling in mind, where a model prescribes the types that will exist in the domain. However, these rules are sometimes too strict and can lead to undesirable anomalies in exploratory modeling scenarios, leading Kühne [77] to propose more relaxed rules for potency reduction in instantiation. This revised form of potency, called "characterization potency", states that the potency of an instance only has to be less than the potency of its type (as opposed to exactly 1 less). This relaxation of the potency rules is not optimal for all domains, however, including the LML deep model for the Process challenge, and is thus part of the justification for less rigid modeling styles and patterns highlighted in **P1** and **P2** in Chapter 1.

As can be seen in Figures 3.3, 3.4, 3.5 and 3.6, every clabject in a deep model possesses a potency value. The postscript integer following the name of a clabject is its potency value. For example, the *Element* clabject in Figure 3.3 has a potency value of 0, which means it is not potent enough to have direct instances of its own. Direct instances of subclasses of *Element* such as *Actor* or *ArtifactType* have to have a *lastUpdated* attribute, however, since these subclasses inherit *lastUpdated*.

Durability: Durability is a property associated with attributes that characterize when instances of a clabject need to have a corresponding attribute. In other words, it characterizes the endurance of attributes over instantiation steps. Like potency, it is represented by a non-negative integer. However, an instance of an attribute must have a durability that is one less than the durability of that attribute. Thus, an instance of a clabject that has an attribute of durability 0 need not have an instance of that attribute.

Because impotent clabjects can represent abstract classes as well as individuals, and the durability of attributes can be any value lower than their clabject potency, it is not possible to formulate a general rule (for all clabjects) governing the relationship between a clabject's potency and durability.

Mutability: Mutability characterizes how the value of an attribute can be changed relative to the value from which it was instantiated, in accordance with the attribute's durability. Like potency and durability, mutability is a non-negative integer value that, with one exception, is always reduced by one when a clabject is instantiated. The exception is when the mutability is already 0, in which case it remains 0. It is an endurance property like durability but measures the endurance of the attribute's value rather than the attribute itself. LML is the only multi-level modeling language that allows the mutability of attribute values to be controlled in this way. If a clabject is an instance of a type with a durable attribute of mutability 0, its corresponding attribute must have exactly the same value. However, if the attribute of the type has a mutability greater than 0, its corresponding attribute can have any value. Therefore a fundamental relationship that must hold between an attribute's durability and its value's mutability is that the mutability must be less than or equal to the durability.

Because of their similarity, durability, and mutability have sometimes been referred to as "forms of potency" in previous papers [84]. However, this introduced an ambiguity about whether the term potency referred to all three or just the original form of clabject potency. Therefore, we do not use this terminology in this thesis, but rather refer to all three properties (potency, durability, and mutability) as "vitalities". The term "potency" therefore refers exclusively to the original form of potency between clabjects.

3.3 Melanee

Melanee is a multi-level modeling tool developed by the Software Engineering Group at the University of Mannheim [14] to support LML using the PLM in a modeling infrastructure based on the OCA. The current prototype implementation is built on the Eclipse platform [114] and uses a plug-in development mechanism allowing users to create their own Eclipse-based modeling tools. This mechanism has allowed Melanee to be extended multiple times since its creation with many different notations, languages, and services.

Among the most noteworthy extensions are custom visualizers [53] that let users define their own domain-specific languages. This includes form-based, text-based, and table-based notations as well as traditional graphical notations, which can be used interchangeably to render a model in different ways for different stakeholders. This is facilitated by “weaving models” that connect a model of the domain content with a model of its representation in a particular concrete syntax. Another noteworthy extension is the so-called “emendation service” which ensures that a Melanee model remains consistent in the face of change [15]. This service checks every change in the model and repairs the affected model elements to make the model consistent again.

The DOCL language presented in this thesis has also been implemented using this plug-in infrastructure.

Chapter 4

Formal Languages

Due to the fact, that DOCL is implemented by using a grammar, to be more precise an ANTLR grammar, to define the syntax of the language this chapter provides a detailed description of the formal foundations of languages and their relationship to grammars and how they can be processed by machines.

Much of the theory in the field of formal languages is based on the work of Chomsky [34], who laid the theoretical foundations for language and grammar classification, and later parsing theory. Any high-level programming language, such as Java [66], has to be translated into machine code that the target computer can execute. This translation process is called *compilation* and has the inputs and outputs shown in Figure 4.1.

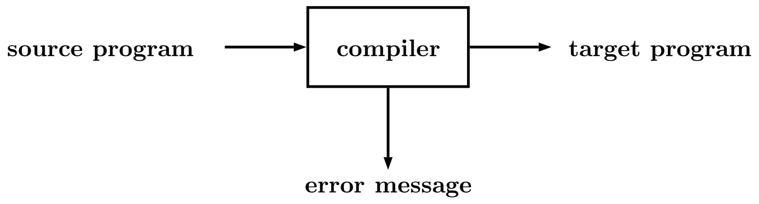


FIGURE 4.1: Compilation Process

If, instead of mapping the source program to a target program, the compiler performs the operations defined in the source directly, then the translation process is called interpreting, and the compiler is called an interpreter.

The overall compilation process is shown in Figure 4.2. First, the input is divided into a stream of characters which are input to the *lexical analyzer*. The lexical analyzer takes groups of characters and tries to classify them into categories called tokens. These are then input to the syntax-directed translation process, which in this case is a parser. A *parser* takes the tokens and arranges them into a hierarchical data structure according to the defined rules of the language. The structure of this so-called *intermediate* representation can vary, but in most cases, it is tree-like [111].

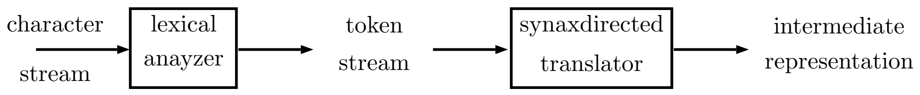


FIGURE 4.2: The steps of a compilation

According to Aho and Ullman [5], three properties have to be specified to create a high-level programming language:

1. the set of symbols that can be used in valid programs,
2. the set of valid programs,
3. the *meaning* of each valid program.

The first of these is relatively easy to achieve. The set of symbols in most modern programming languages is a mixture of the English alphabet and arithmetic operation symbols. However, it is much more difficult to define the set of valid programs [5]. When specifying a programming language, grammatical rules can be used to reduce the size of the set of valid programs. However, they may evaluate statements such as the one in Listing 4.1 as valid [5]. This kind of statement will either result in an endless loop or, if the meaning of such a valid program is not defined, in some kind of error due to the consequences of the third property.

```
L GOTO L
```

LISTING 4.1: Potentially valid *FORTRAN* statement

The third and final property states that a programming language has to determine the semantics of a valid program. Listing 4.1 therefore should ideally be rejected on the grounds that it is ambiguous because if executed it will enter an infinite loop. Either way, the language specification has to be able to determine the semantics of a valid expression [5].

The words λ, 01110,01,00010,0,1 are words over the alphabet $\Sigma = \{0, 1\}$ (λ being the empty word). The set of all words over an alphabet is denoted by Σ^* and by Σ^+ for all non-empty words. These two sets are infinite for any Σ, and they are the free monoids and free semigroup generated by Σ [108]. If a language L is infinite it is not possible to enumerate all possible combinations of words in order to translate the language into executable code. Thus, we have to look for another representation of the translation. This specification of a language has to be of finite size, but the language specified is not required to be finite [5].

There are two well-known methods to fulfill this requirement. The first method is to define a generative system, usually called a *grammar*. The second method is to define a *recognizer* which when presented with a finite string, answers "yes" if the string is part of the language and "no" if the string is not part of the language [5]. The binary output of a recognizer is not sufficient for this purpose. A grammar offers much more than just returning a Boolean value, it can also generate statements in that language. Tools for grammar systems support the automatic generation of parse trees and much more. The following sections explain the first method in detail because this approach was used to implement the DOCL language developed in this thesis.

4.1 Theory

The process of language engineering is defined in terms of alphabets, words, languages, and their grammars. There are multiple kinds of grammars that can be classified depending on their properties. These properties are also transmitted to their respective languages.

This section defines all of the formal parts that are needed to create a (programming) language.

Definition 4.1 (Alphabet)**.** An *alphabet* is a finite nonempty set. The elements of an alphabet Σ are called *letters* or *symbols*[108]

Definition 4.2 (Word)**.** Let $\Sigma = \{a, b, c, ...\}$ be a set of symbols, or *alphabet*, then a word w over Σ is a string where each character from w is from Σ. $w = a_0, a_1, a_2, ..., a_n$ where $a_i \epsilon \Sigma$. A special word is the "empty" word ε.

Definition 4.3 (Length of a word)**.** The length of a word is displayed by $|w|$ where $w = a_1 a_2 a_3 ... a_n, |w| = n$ and $|\varepsilon| = 0$.

The concatenation of two words is defined as follows.

Definition 4.4 (Concatenation)**.** Let $w_1 = a_0 a_1 a_2 ... a_n$ and $w_2 = b_0 b_1 b_2 ... b_n$ two different words where $a_i b_i \epsilon \Sigma$, then

$$w_1 \circ w_2 = a_0 a_1 a_2 ... a_n b_0 b_1 b_2 ... b_n$$

is the concatenation of w_1 and w_2.

Definition 4.5 (Language)**.** Let Σ^* be a set of words, then a language L is a subset of Σ^*

$$L \subseteq \Sigma^*$$

Definition 4.6 (Grammar)**.** A Grammar is a tuple $G = (N, \Sigma, P, S)$ where

(1) N and Σ are alphabets, $N \cap \Sigma = \emptyset$

(2) $S \in N$. (The elements of N are called *nonterminals* and those of Σ *terminals*)

(3) S is called the *start symbol*.

(4) P is a finite subset of

$$(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$$

Definition 4.7 (Regular grammar)**.** A grammar is called *regular* if each production is of the form $u\xi v \to uyv$, where ξ is in $N - \Sigma$, u and v are in $(N - \Sigma)^*$, and y is in $N^* - \{\varepsilon\}$. The language generated by a "regular grammar" is called a "regular language".

A regular grammar or regular language is also called context-sensitive [54]. This means that the grammar is expressed using rules of the form $u\xi v = uyv$ where ξ can be rewritten to y. The equivalent automaton that accepts regular grammars is called a *finite automaton* [61].

Definition 4.8 (Context-free grammar)**.** A grammar is called *context-free* $G = (N, \Sigma, P, S)$ if each production in P is of the form $\xi \to n$, where ξ is in $N - \Sigma$ and n is in N^*. $L \subseteq \Sigma^*$ is said to be a *context-free language* if and only if L is generated by some context-free grammar.

In practice, the term context-free means that the results of the productions are rewritten independently of the context in which each variable appears. Every context-free grammar is also context-sensitive, but the converse is not true [54].

Like regular grammars, the set of context-free grammars also has an equivalent automaton which is called a *pushdown automaton*. This automaton accepts all context-free grammar definitions. A pushdown automaton is deterministic if the following restrictions hold [61]

1. whenever $\delta(q, a, X)$ is nonempty for some a in Σ, then $\delta(q, \epsilon, X)$ is empty,

2. for each q in Q, a in $\Sigma \cup \{\epsilon\}$ and X in Γ, $\delta(q, a, X)$ contains at most one element.

The first restriction prevents the pushdown automaton from choosing either the next input or making an ϵ-move. The second restriction prevents a choice on the same input [61].

Definition 4.9 (Pushdown automaton)**.** A finite deterministic automaton can be defined as a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where

- Q is a set of states,
- Σ is an alphabet called *input alphabet*,
- Γ is an alphabet called the *stack alphabet*,
- q_0 in Q is the initial state,
- Z_0 in Γ is a particular stack symbol called the *start symbol*,
- $F \subseteq Q$ is the set of final states,
- δ is a mapping from $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ to finite subsets of $Q \times \Gamma^*$.

To sum up these definitions, first, the alphabet of a language was defined, which is a set of arbitrary symbols from which words can be formed. The definition also provides a way to measure the length of a word and to concatenate two or more words into a new word. Second, grammars were defined and how they relate to languages. A subset of grammars are context-free grammars that generate context-free languages and are accepted by pushdown automatons. "Context-free grammars are a generalization of regular grammars in that no restrictions are placed on the right-hand sides of rules." [111]

By definition, every context-free language is generated by a context-free grammar [94]. An element (α, β) in P will be written $\langle\alpha\rangle \rightarrow \langle\beta\rangle$ and is called a production, where α is the left-hand side and β the right-hand side of the production. In other words, a grammar is context-free if the finite set of rules or productions has only non-terminals on the left-hand side of a production and an arbitrary number of combinations of terminals and non-terminals on the right-hand side of a production. The left-hand side of a production is also called a *syntactic category*, and every category itself represents a language [61]. The deterministic version of the pushdown automaton accepts only a subset of all context-free grammars. This subset includes the syntax description of most programming languages [61].

4.2 Lexical Analysis

In a compiler or interpreter, the first step of processing an expression is the "linear analysis", also called the "lexical analysis". This step aims to categorize each token or a group of tokens. Usually, this is done from left to right. Assume the following statement is input to the translation process:

```
position := initial + rate * 60
```

LISTING 4.2: Input for the lexical analysis

The lexical analysis would group the expression into the following token categories:

1. the identifier `position`,
2. the assignment symbol `:=`,
3. the identifier `initial`,
4. the plus sign,
5. the identifier `rate`,
6. the multiplication sign,
7. the number 60.

The blanks between the tokens are usually eliminated during the lexical analysis [6]. If the result of the parsing process is in fact a parse tree then every token or classification is a leaf in the hierarchical parse tree. The following sections show how these rules can be organized in such data structures.

4.3 Syntax and Semantics

Any language has some grammatical structure which consists of a set of rules on how words form a sentence or how words in a sentence relate to each other. For a natural language, e.g. English, every word of a sentence can be labeled or classified into syntactic categories. The sentence -

The pig is in the pen.

can be represented as a labeled tree, as presented in Figure 4.3.

This representation helps clarify the overall structure of the English language and how sentences are properly composed. The same principle can be

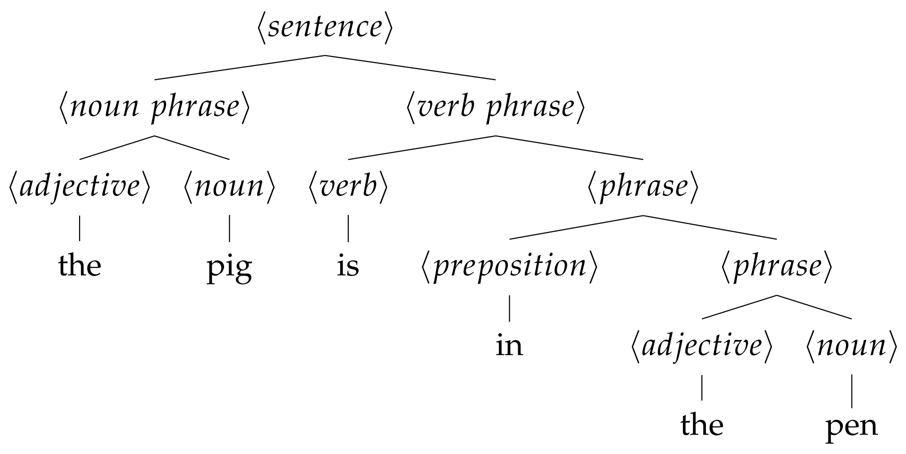


FIGURE 4.3: Tree structure of a sentence in English [5]

applied to a programming language or any other language for that matter. For example, the arithmetic expression

$$a + b * c \tag{4.1}$$

can be also represented as a labeled tree, just like a sentence from a natural language. The labels may differ, but the tree helps to understand the relationship between the symbols in that particular language [5]. In figure 4.4, the labeled syntax tree is shown for an arithmetic expression. This example shows a calculation order in which the multiplication is computed before adding the left part of the tree to the result. So in fact this syntax tree ensures the correct calculation of an arithmetic expression because the right term, the multiplication, has to be resolved before the result is added to a.

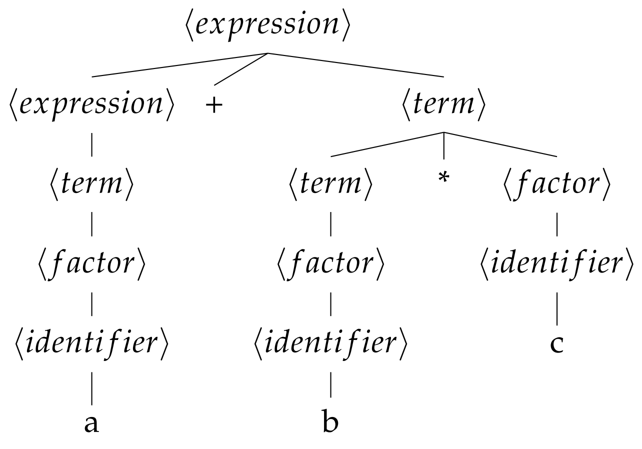


FIGURE 4.4: Tree structure of an arithmetic expression [5]

The example that was shown earlier in Listing 4.2 can also be represented as a parse tree, which is shown in figure 4.5. This example also shows that some logical or arithmetic expressions have to be resolved before other expressions. The phrase `rate * 60` is a logical unit because arithmetic rules say that multiplication is performed before addition. The rules that create such a hierarchical structure might have the following form:

1. any `identifier` is an `expression`,
2. any `number` is an `expression`,
3. if $\texttt{expression}_1$ and $\texttt{expression}_2$ are `expressions`, then so are

$$\texttt{expression}_1 + \texttt{expression}_2$$
$$\texttt{expression}_1 * \texttt{expression}_2$$
$$(\texttt{expression}_1)$$

.

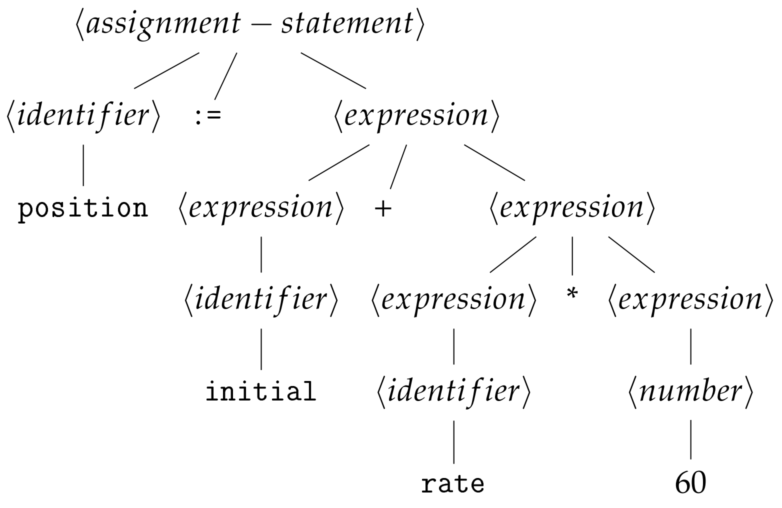


FIGURE 4.5: Parse tree for *position := initial + rate * 60*

The first two rules are non-recursive rules whereas the third rule defines a set of two expressions with different operators applied to the two expressions [6]. Thus, as defined by the first rule, `initial` and `rate` are `identifiers`. The `number` 60 is an expression, which is defined by the second rule. The third rule first matches `rate * 60`, which is an expression, and then to `initial + rate * 60` which is also an expression itself.

Figure 4.6 shows three parse trees that are non-recursive (i), left-recursive (ii), and right-recursive (iii). "We say that φ *dominates* $\psi (\varphi \rightarrow \psi)$ if there is a derivation $\sigma_1, ..., \sigma_n$ such that $\sigma_1 = \varphi$ and $\sigma_n = \psi$ (i.e., if ψ is a step of a σ-derivation)." [33] With this definition by Chomsky, it can be shown that A in (i) is nonrecursive for non-null φ, ψ, because of $A \Rightarrow \varphi A \psi$. A in (ii)

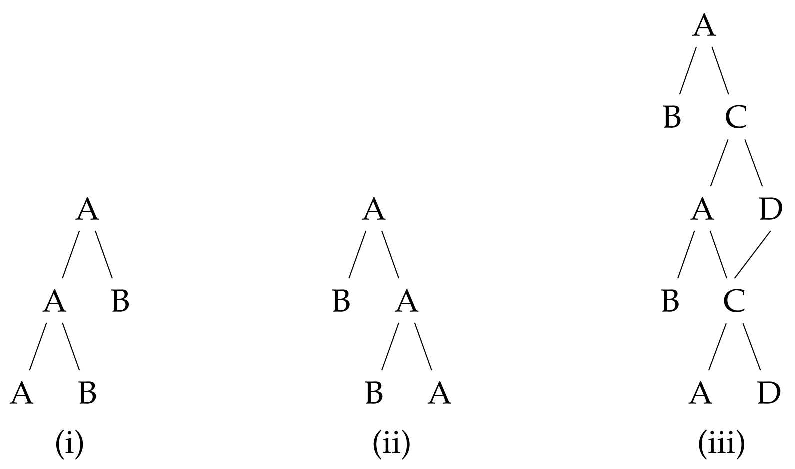


FIGURE 4.6: Example parse tree for non-, left- and right-recursion [33]

is left-recursive if there is a non-null φ such that $A \Rightarrow A\varphi$ and A in (iii) is right-recursive if there is a non-null φ such that $A \Rightarrow \varphi A$ [33].

4.4 Parsing Strategies

The parser obtains a stream of tokens from the lexical analyzer and checks if the whole stream can be generated by the defined grammar for the source language. The role of the parser in the compiling process is shown in figure 4.7.

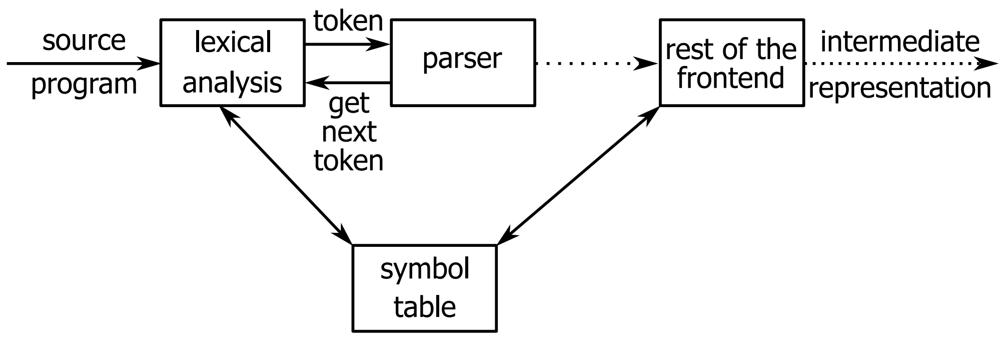


FIGURE 4.7: The role of a parser in the compalitation process

There are two types of algorithms that can be used for the task of syntactic analysis: top-down and bottom-up [65]. The top-down parser method builds parse trees from the top (root) to the bottom (leaves). The bottom-up parser method begins with the bottom (leaves) and works its way up to the top of the tree (root). The input for both parser methods is the same and is scanned from left to right one symbol at a time and is processed. These parsing strategies only accept subclasses of grammars, such as LL and LR grammars [6].

The class of context-free grammars that can be parsed deterministically in a top-down fashion is called *LL(k)*, and the class of context-free grammar that can be parsed deterministically in a bottom-up fashion is called *LR(k)* [107].

With the definition of a context-free grammar (cf. definition 4.8) in mind, a grammar with LL(k) properties can be defined as follows.

Definition 4.10 (LL(k) grammar)**.** A grammar $G = (N, \Sigma, P, S)$ is said to be a LL(k) grammar for some positive integer k if and only if given

1. a word w in N^* such that $|w| \leq k$,
2. a non-terminal A in Σ,
3. a word w_1 in Σ^* ,

there is at most one production p in P such that for some w_2 and w_3 in N

4. $S \Rightarrow w_1 A w_3$,
5. $A \Rightarrow w_2$,
6. $(w_2 w_3)/k = w$.

According to Rosenkrantz and Stearns [107], "*LL(k)* grammar is a context-free grammar, such that for any words in its language, each production in its derivation can be identified with certainty by inspecting the word from its beginning (left end) to the *k-th* symbol beyond the beginning of the production. Thus when a non-terminal is to be expanded during a top-down parse, the portion of the input string that has been processed so far plus the next *k* input symbols determine which production must be used for the non-terminal."

Any context-free language that is generated by a *LL(k)* grammar can be recognized by a deterministic push-down automaton [86]. Every *LL(k)* grammar is also an *LR(k)* grammar [75].

In the past, when the resources in a computer were limited, programmers had to write their grammars for deterministic parser generators. Today, however, programmers are able to use non-deterministic parsing strategies since the processing resources in computers have grown [99]. These strategies are called Generalized LR (GLR) and Generalized LL (GLL), and they are able to handle nondeterministic ambiguous grammars. A grammar that is ambiguous, returns multiple parse trees (forests) because they were intentionally designed for natural languages [99].

4.5 ANTLR

One of the most popular and widely used tools for parsing structured text or source code is ANTLR, which stands for "ANother Tool for Language Recognition". It is a language-agnostic parser generator that can generate top-down parsers for multiple programming languages, including Java, C#, Python, and others [99]. This section provides an overview of ANTLR's properties since this is the technology used in this thesis to realize the developed deep object constraint language.

The top-down strategy used by ANTLR is called *ALL(*)* [99]. This strategy can also handle nondeterministic and ambiguous grammars but does not return multiple parse trees, like GLL or GLR. The *LL* parsing style pauses at each production until the prediction mechanism has chosen the correct production to expand the tree and resumes the parsing process. The *ALL(*)* strategy parses the whole expression dynamically. At each decision point in the grammar, multiple sub-parsers are launched. For every possible decision at a particular point in the grammar, one parser is created that tries to match the input. If the path a sub-parser has been taking fails to match the input, it dies off and is no longer considered a valid production.

ANTLR also generates lexers in combination with recursive decent parsers from the grammar definition so that no other component has to be generated to recognize expressions in that grammar. Listing 4.3 shows a grammar with left-recursive rules such as `expr`. Such rules are unacceptable in ANTLR3 but ANTLR 4 can handle them by automatically rewriting the grammar into a non-left-recursive and unambiguous grammar. Another reason why this grammar cannot be accepted by ANTLR 3 is that the *stat* rule has alternative productions that have a common prefix(i.e., *expr*). This rule is undecidable for ANTLR 3 and for *LL(*)* style grammars [99].

```
grammar Ex;
// action defines ExParser member: enum_is_keyword
@members {boolean enum_is_keyword = true;}
stat: expr '=' expr ';' // production 1
    | expr ';' // production 2
;
expr: expr '*' expr
  | expr '+' expr
  | expr '(' expr ')' // f(x)
  | id
;
id : ID | {!enum_is_keyword}? 'enum' ;
ID : [A-Za-z]+ ; // match id with upper, lowercase
WS : [ \t\r\n]+ -> skip ; // ignore whitespace
```

LISTING 4.3: An example of a left-recursive grammar from Parr et al.[99]

In theory, the *ALL(*)* parsing strategy has a runtime complexity of $O(n^4)$, and the GLL parsing strategy has a runtime complexity of $O(n^3)$. Nevertheless, when using this strategy for expressions in common languages, Parr et al. [99] showed that *ALL(*)* parsers exhibit a linear behavior and complete the parsing process faster than implementations of GLL parsing strategies.

Before ANTLR4 generates the parser, it rewrites all direct left-recursion rules.

The grammar for DOCL is a context-free left-recursive grammar that uses the *ALL(*)* parsing technology.

Part III

DOCL

The third part of the thesis presents the DOCL language developed in this thesis. The first chapter describes the idea behind reflective constraints, which are one of the most novel features of the language, and explains how the notion applies in the context of deep modeling. The second chapter presents the concrete new features of the DOCL language. Finally, the third chapter provides an overview of how the prototype implementation of the language is realized as a plugin to the Melanee tool.

Chapter 5

Reflective Constraints in Deep Modeling

Software engineering, and indeed IT in general, revolves around the creation of descriptions of things expressed in formal languages. Formal languages are important since they allow descriptions to be checked for adherence to syntax rules and algorithmically processed. An important example is programming languages which are designed to support the description of programs that, when transformed and executed, can instruct a computer to take certain actions. Another example are database schema languages which are designed to support the description of schemas that control how data about a particular subject is stored in a database management system. The distinction between a language and a description of something represented in that language also occurs when modeling languages are used to create descriptions (i.e., models) of specific subjects.

In the case of programming languages, the nature of the language used to represent a program is usually not of concern at execution time when that program is actually deployed in its operation environment to direct the behavior of that environment, but it is important for the compilers and/or virtual machines that make the execution of programs possible. However, for applications where a high level of behavioral flexibility is required, it is often advantageous to let a running program be "aware of" its description so that it (the program) can reason about, and potentially manipulate, that description (i.e., "itself") at execution time. This capability is generally called reflection, but there is no consensus on precisely what this term means. In this chapter, we clarify the terminology used in this thesis in the specific context of multi-level, object-oriented modeling.

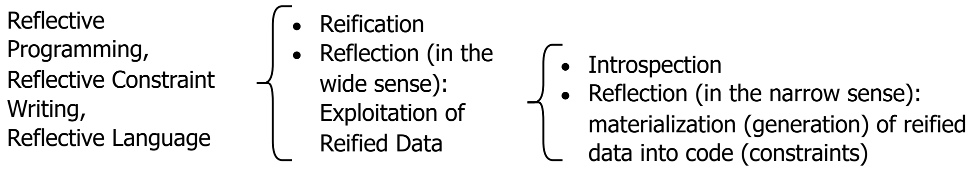

Reflective Programming, Reflective Constraint Writing, Reflective Language
• Reification
• Reflection (in the wide sense): Exploitation of Reified Data
• Introspection
• Reflection (in the narrow sense): materialization (generation) of reified data into code (constraints)

FIGURE 5.1: Reflective programming terminology and its application to reflective constraint writing defined by [48]

5.1 Terminology

In the programming language domain, the term reflection is used in both a narrow sense and a wide sense [48]. The wide sense covers any capability or technology that allows a program to gain access to a description of itself at application time (i.e., run time). For example, Java offers the Java reflection API. This "access" can take place at two basic levels, however. One form of access, the more limited one, occurs when a program can view, and reason about, a representation of itself but not change it. This is often called *introspection* [90]. The other form of access, the more powerful one, occurs when a program is not only able to view a representation of itself but also to change that representation at run-time. In other words, it can change its own description while it is being applied (i.e., executed) at run time. This is sometimes called *intercession* [90].

Unfortunately, the term reflection is also often used to refer just to intercession to provide a contrast to introspection, rather than, or in addition to, the general notion of a program exploiting its description at run time. Draheim refers to the use of reflection to mean intercession as *reflection in the narrow sense* and the more general form of reflection as *reflection in the wider sense* [48]. To avoid confusion, in this thesis, we use the term only in the *reflection in the wider sense*, and we use the term intercession when we want to refer to *reflection in the narrow sense*.

As mentioned in Figure 5.1, a fundamental prerequisite for reflection is that a program has access to a representation of itself at run time. This means that a program needs to be able to access (for introspection) and change (for intercession) a representation of its description at run time. Since run-time information in object-oriented systems is, by definition, represented as objects (described by classes), this means that the aforementioned description itself needs to be described and represented in an object-oriented way. This process of turning a description into an object-oriented form is generally referred to as *reification*. Reification is, therefore, a prerequisite for reflection in object-oriented technologies, since it allows a program to access and manipulate a description of itself in the form of objects.

In general, the features of a language are distinct from the features of specific subjects that are described by that language. For example, a general-purpose, object-modeling language, which offers description concepts such as classes, attributes, and associations, can be used to describe unlimited numbers of other languages such as use case diagrams, sequence diagrams, activity diagrams, etc., or other subjects such as systems or physical objects, etc. However, unlike most languages, such a general-purpose object-modeling language can also be used to describe "itself" – that is, a language that offers concepts such as classes, attributes, and associations. The occurrence of overlaps between the constructs supported by a language and the constructs described in a particular usage of that language is often referred to as a *meta-circularity*. This can be exploited in the context of a programming language to support reflection and the creation of environments for the language that is self-bootstrapping [35].

5.2 Reflection in Object-Oriented Modeling

In object-oriented modeling, which is the subject of this thesis, subjects of interest are conceptualized using objects along with their links and properties and are described as object models in terms of concepts such as classes, associations, and attributes. Such object models are usually composed of two parts – a core (often graphical) model, usually called a class diagram, which describes the interlinked objects, and an auxiliary model which describes additional properties of these objects, usually in a textual form. These two parts are typically written in distinct languages that are members of a larger suite of modeling languages. As explained in Chapter 2, in the case of the MDA technology space supported by the OMG, the UML/MOF is usually used to represent the core model, and the OCL is used to represent the auxiliary model. In the Epsilon technology space, on the other hand, the UML is used for the core model while the EOL is used use for the auxiliary model.

In a traditional two-level modeling context, the degree to which such languages support reflection varies depending on the extent to which the auxiliary language can be regarded as an action language – that is, a language that can change as well as access objects, links and slots in the subject of the model. The reification of descriptions (i.e., models), which is a fundamental prerequisite for reification, is automatically satisfied by object-oriented models since they are inherently represented as objects and are thus amenable to application-time manipulation. However, reification is only possible if the auxiliary language supports the required features. More specifically, if the auxiliary language provides features for accessing the (reified) representation of a model at application time, introspection is possible, and if further features are available for changing the model, intercession is also possible.

5.2.1 Reflection in OCL in a Two-level Context

In the traditional two-level modeling setting for which it was developed, the OCL provides limited support for reflection.

Introspection: Since it is intended to be a declarative language that does not make changes to the modeled (i.e., described) objects at application time, OCL's reflection features are primarily focused on introspection. These are summarised in Figure 5.1. The first category of operations allows properties of objects, at the instance level, to be queried in terms of concepts in their description (i.e., primarily their class/type and their associated state model). The second category allows the properties of a type to be queried at the instance level, as part of a constraint. For example, it is possible to determine what features (attributes, associations), what supertypes, and what instances a type has.

- Properties of all objects, i.e., objects $o : OCLAny$:
  - o.oclIsTypeOf(t:OclType):Boolean – *true iff* $o : t \wedge \nexists t'.t < t'$
  - $o.o$.oclIsKindOf(t:OclType):Boolean – *true iff* $o : t$
  - o.oclIsInState(s:OclState):Boolean – *test for machine state*
  - o.oclIsNew():Boolean – *postcondition test for object creation*
  - o.oclClassType(t:OclType):instance of Classifier – *type casting operation*
- Properties of meta objects *t: OclType* representing user-defined types:
  - t.name:String – *the name of the type t*
  - t.attributes:Set(String) – *the set of names of the attributes of t*
  - t.associationEnds:Set(String) – *names of association ends navigable from t*

- *t*.operations:Set(String) – *the names of the operations of t*
- *t*.supertypes:Set(OclType) – *the set of all direct supertypes of t*
- *t*.allSupertypes:Set(OclType) – *the set of all supertypes of t*
- *t*.allInstances:Set(type) – *the set of all instances of type t*

Using these operations, OCL expressions can access information about an object model to influence the results they deliver at application time. When used for the purpose of defining constraints, for which OCL was designed, these "results" are the outcomes of conformance checks (of a set of objects described by a model). However, OCL is often used as the navigation sub-language of other languages. For example, the ATL transformation language uses OCL to navigate to sets of objects in the input models and describe what sets of objects they are mapped to in the output. In this context, the "results" of the application of the OCL expressions are the generated transformation.

Intercession: To support intercession, a language needs to allow a system to change its description at application time. OCL potentially supports a very limited form of intercession in a two-level context by means of "definition" constraints. An OCL definition constraint can declare that a class in a UML model has a new attribute, with a specified value, or a new operation, with a specified body. Thus, OCL constraints defined on a UML model can theoretically extend that model when applied in the context of instances of that model. However, for two reasons this is extremely limited -

1. It is unclear what mechanism would trigger the application of these constraints in a particular use of the model to control the structure of a set of objects. Normal OCL constraints are usually triggered when changes are made at the instance level (i.e., to objects, links, and slots in the described model) to check their validity. However, since OCL definition expressions change the model itself, when should they be triggered?

2. Even when an OCL definition expression is executed in order to extend a UML model, there is virtually no way for OCL constraints defined on, and applied to, an instance of that model to exploit the new information. As shown in the figure above, there are no introspection operations an OCL expression can use to access any new operations, and while there is an operation to access new attributes, only the name of the attribute is returned not the value. So the intercession features of standard OCL, to the extent that they exist, are almost useless.

5.3 Reflection in Deep Modeling

As mentioned above, since object models can be the subject of other object models, it is useful to create hierarchies in which more concrete and specific models are described by more abstract and general models. Such model stacks can provide richer and more subtle opportunities for supporting reification.

5.3.1 Linear Model Stacks

The most well-known example of the use of this "meta modeling" approach is the veritable four-layer modeling hierarchy at the heart of the OMG's MDA technology space (see Figure 2.2). This pioneered the use of meta-models (at level 2) and meta-meta-models (at level 3) to define languages which are then used to model other subjects. Two of the most well-known and important languages in model-driven development are defined and accommodated in this stack - the UML, which is regarded as occupying, and being defined by, level 2 of this hierarchy, and the MOF which is regarded as occupying and being defined by level 3 of this hierarchy. In short, the UML is defined in terms of the MOF.

An obvious question is how the hierarchy of descriptions is terminated. In other words, in the context of the MDA infrastructure, what language is used for defining the MOF? Since the MOF was optimized for describing modeling languages, it is also optimized for, and capable of, defining itself. This situation could easily be represented by adding another level, level 4, above level 3, which has exactly the same contents (i.e., the MOF). But this does not answer the termination question, it just shifts it up a level.

To avoid defining an infinite hierarchy of model levels, each defined by the level above, it is convenient to exploit the aforementioned notion of meta-circularity and regard the MOF as being defined in terms of itself. The OMG infrastructure not only does this, but it also goes further, since one of the sublanguages of the UML is a general-purpose modeling language which is essentially the same as the MOF. When defining the 2.0 versions of the UML and MOF, the OMG, therefore, exploited meta-circularity to remove all residual differences between the two and merge them into a single unified modeling language. At the heart of MOF 2.0 and UML 2.0 is a unified common core set of modeling features that they share through the so-called "Infrastructure Library". By means of meta-circularity, therefore, the UML both contains (as a sublanguage) the common core and is defined in terms of (as a metalanguage) the common core. In other words, the UML is defined in terms of a subset of itself, which is called the MOF.

As pointed out by Clarke, when constraints "[...] are associated with meta-classes, the conditions apply to classes, this provides a way of expressing structural and behavioral semantics for a language" [35]. The OMG language specifications make good use of this fact by extensively using OCL constraints to define the precise rules governing the use of the language being specified. This includes the definition of the UML, which thanks to the aforementioned meta-circularity can use OCL constraints defined in the context of the common MOF core to govern the structure of the definition of the UML itself, and also to govern the structure of the other sublanguages of the UML.

Technically, the degree to which a standard constraint language, defined for a two-level modeling context, supports reflection does not change when it is used in a multi-level modeling context. However, for languages that support intercession (i.e., action languages like EOL) the potential impact of reflection is greatly amplified if there is a metacircularity at the top level of the stack. This is because intercession expressions applied at the second level can change the nature of the top level, and thus potentially change the most fundamental aspects of the modeling approach used in the whole stack. This is why tools like XModeller^{ML}[38] can provide such fundamental flexibility and allow programs to change almost every aspect of the execution environment while they are running. While this is extremely powerful, it is also extremely dangerous. The value of introspection is not amplified in the same way when a standard constraint language is used in a multi-level model stack since expressions in the language can still only gain access to the level above.

5.3.2 Orthogonal Classification Architecture

As explained in chapter 3.1.1, most modern multi-level modeling approaches are based on the OCA rather than on a strictly linear model stack. This organizes classification relationships into two orthogonal dimensions in which a single linguistic model classifies (and thus spans) multiple ontological dimensions arranged in a linear stack. This modeling architecture has a big influence on the potential impact of introspective capabilities as well as intercession capabilities when applied in the context of a two-level constraint language. This is because the linguistic dimension is essentially composed of two levels, the linguistic (meta) model and the stack of ontological levels, so the reflection capabilities can affect the whole ontological stack even though there are only two levels.

The OCA also means that a meta-circularity at the top of the linear ontological model stack because (a) there are no questions about what language the top level is defined using (it is the linguistic (meta)model) and (b) if intercession features are available, the ability to use them to change fundamental aspects of the modeling approach is already possible by modifying the linguistic (meta)model. This allows the higher levels of the ontological stack to focus on modeling abstract concepts in the domain of interest rather than on defining general modeling language concepts. The only meta-circularity that exists in an OCA-based model, therefore, is the meta-circularity at the top of the linguistic model stack which explains what the linguistic (meta)model is defined using (i.e., itself).

Introspection

As mentioned above, the OCA significantly boosts the power of constraint languages because the full stack of ontological models lies within the constraining power of a constraint language. Thus, even a standard OCL constraint that is not multi-level aware can be used to significantly constrain the nature of the modeling rules used to populate and relate the ontological levels. This power is enhanced even further when a multi-level aware constraint language with reflective capabilities, like DOCL, is used as explained in this thesis.

The OCA's separation of classification relationships into two orthogonal dimensions naturally gives rise to linguistic and ontological forms of both kinds of reflection.

Linguistic introspection: Introspection into the linguistic (meta)model significantly increases the utility of even limited introspection capabilities like those of standard OCL. This is because they allow information about the whole ontological model stack to be queried, and thus allow reasoning and decision-making on a multi-level basis. In principle, standard OCL could be used to extract detailed information based on the structure of the ontological model, but because it is not designed with multi-level modeling in mind, quite complex expressions are usually required. A multi-level aware

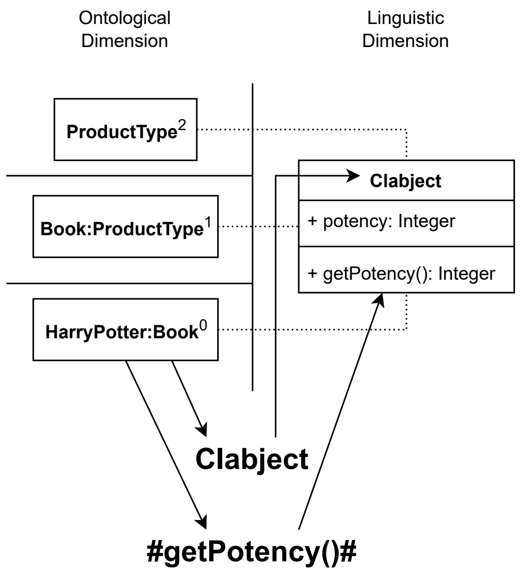


FIGURE 5.2: Linguistic Introspection

constraint language can provide multiple features that support linguistic introspection over a multi-level model in a much simpler and intuitive way. Figure 5.2 shows an example of linguistic introspection.

```
Clabject -> not (exists(clabject|clabject.#getPotency()# = -1))
```

CONSTRAINT 5.1: Linguistic Introspection Expression

Constraint 5.1 begins with a query to retrieve all clabjects in the model. This constraint must therefore hold for all clabjects that are already in existence and for all clabjects that might come into existence in the future. For this kind of query, we can use all classes that exist in the PLM meta-model of Melanee. It is also possible to substitute *Clabject* with *Level* to get information about a particular level.

Ontologial introspection: Introspection into an ontological (meta)model is, in principle, completely symmetric to linguistic introspection since in the OCA there is no asymmetry between ontological and linguistic types of a clabject. However, if a language that is unaware of the multiple levels is used to define expressions, like standard OCL, direct introspection of the kind shown in Figure 5.2 is not possible. This is because standard OCL cannot recognize ontological types of a clabject since an OCA environment implemented in a traditional modeling infrastructure includes classification (i.e., instanceOf) relationships as normal associations. When using standard OCL in an OCA context, the introspection operations defined in Figure 5.2 operate

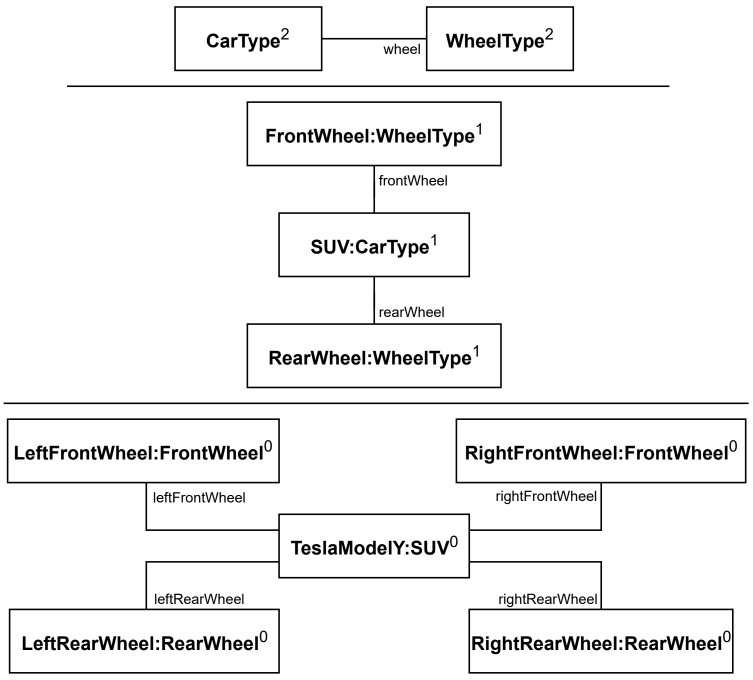


FIGURE 5.3: Ontologocigal introspection

exclusively in the linguistic dimension and cannot be used to query information from the ontological types of a model element.

One of the main roles of a multi-level aware constraint language in an OCA context is therefore to support ontological as well as linguistic reflection in a symmetric way, using symmetric sets of operations.

```
context TeslaModelY(2,2)
inv: self.$CarType$.wheels -> includesAll(Set{leftFrontWheel,rightFrontWheel,
    leftRearWheel,rightRearWheel})
```

CONSTRAINT 5.2: Ontological Introspection Constraint

In Figure 5.3, a deep model is shown where the *TeslaModelY* clabject is at the lowest level and is connected to four wheels. The ontological introspection feature of DOCL allows the constraint to navigate the classification hierarchy in the upward direction. The "$" symbol represents this navigation and searches for the type or, in this case, the deep type specified. The expression can then use navigations of that type or access attributes or methods. In Constraint 5.2, the *wheels* navigation is used and when the expression ends the wheel instances (from the level of the context) are returned.

Intercession

Distinguishing between ontological and linguistic intercession in an OCA context does not really extend the power of the intercession capabilities supported by constraint languages because intercession operations operating on the (meta-circular) top-level of a linear model stack can, in principle, already change almost all modeling mechanisms in that stack. However, their separation can significantly reduce risks and ambiguities as mentioned below.

Linguistic intercession: Intercession into the linguistic (meta)model) occurs when constraint language expressions executed in the ontological dimension of an OCA-based model make changes to the linguistic metamodel. As mentioned previously, this is only possible with languages like EOL, that offer action language capabilities. Since the linguistic metamodel defines the fundamental concepts and rules by which information in the ontological stack is modeled, this can have major consequences on the semantics and structure of models and the modeled information (i.e., on the representation of the subject). However, because ontological information is strictly separated from linguistic information, modelers who exploit linguistic intercession should be (a) fully aware of the risks of manipulating the linguistic metamodel at application time and (b) able to do so without the clutter of ontological information.

Ontological intercession: Intercession into an ontological (meta)model) occurs when constraint language expressions executed conceptually at one ontological level of an OCA-based model make changes to one or more classifying levels above them in the ontological stack. Because the OCA allows ontological models to focus on representing ontological concepts, usually without the need for any kind of meta-circularity at the top of the stack, expressions in a constraint language can exploit ontological intercession in a simpler and lower-risk way since there is no danger of accidentally affecting fundamental, pan-level modeling features. This is important since most motivations for, and uses of, intercession are for ontological purposes (i.e., to access and manipulate concepts related to an application of the modeling infrastructure to a particular subject) rather than linguistic (i.e., to access and manipulate the fundamental concepts and rules use to construct models).

Chapter 6

DOCL Features

This chapter presents an overview of the features provided by DOCL to support the introspection mechanisms described in the previous chapter. Since DOCL is a conservative extension to OCL, all but four esoteric features of OCL are also supported by DOCL. Given the size of the language, the supported features of OCL are not described again here for space reasons – only the four deprecated features are mentioned in the final subsection of this chapter. The rest of the chapter presents the abstract syntax, concrete syntax, and semantics of the new DOCL features that are not included in standard OCL.

To illustrate the new features, we use the running example from the process challenge introduced in Chapter 3. The linguistic and ontological navigation feature builds on a master thesis that developed a forerunner of DOCL [71].

6.1 Linguistic Introspection

Navigation over the linguistic dimension is one of the most important (and frequently used) features in DOCL. It enables access to the linguistic meta-level, thus facilitating the reflective capabilities described in the previous chapter. For example, clabjects can be queried for their linguistic properties, such as their vitality properties, or their place in the classification or inheritance hierarchies. Not only clabjects can be queried, every kind of element in a model can access its linguistic meta-model properties.

From a language usability perspective, the syntax and semantics of the linguistic introspection features have to be unambiguous and clear.

```
context ProcessType (2,2)
inv: self.#getPotency()# = 0
```

CONSTRAINT 6.1: Linguistic Introspection Expression in DOCL

DOCL uses a special symbol to designate a switch to the linguistic dimension of a deep model element. This approach is as unambiguous as possible and avoids collisions with the existing OCL syntax for identifying normal attributes and methods. OCL solved the problem of avoiding name clashes by prefixing the names of "special" operations providing some kind of introspection capability with the string "ocl-". A good example is the *oclIsTypeOf()* operation whose name is highly unlikely to collide with user-defined method names and is usable without needing any special symbols. DOCL supports this form of introspection as well, but DOCL has to differentiate between the two kinds of classification (ontological and linguistic) as well. It is possible to create a *Method* in a clabject that is called "getPotency()", which occupies the ontological dimension, and despite the fact that there exists the linguistic operation "getPotency()" as well. In DOCL, the '#' symbol is used to enclose linguistic navigation expressions. As shown in Constraint 6.1, this makes it clear exactly where the linguistic navigation begins and where it ends. The example DOCL constraint is an invariant which makes sure that every deep direct instance of *ProcessType* at level O2 has a potency value of 0. After the *getPotency()* operation, the context switches back to the ontological dimension.

DOCL performs just one dimension switch at a time and immediately switches back to the ontological dimension after the linguistic navigation has been resolved. That means if the user wants to navigate in the linguistic context twice, each expression has to be enclosed by '#'. Constraint 6.2 shows an example of such expressions applied to the *O0* level of the running example shown in Figure 3.3. After the linguistic navigation ends, the user can navigate and perform collection operations again on the ontological level. This syntax for linguistic navigation is precise and leaves no room for ambiguities.

```
context TaskType(1,_)
inv expectedDuration: self.#getDirectSupertype()#.#getSubtypes()# -> reject(
    self) -> forAll(t|t.expectedDuration < self.expectedDuration)
```

CONSTRAINT 6.2: Linguistic Introspection in a DeepModel

The second navigation in Constraint 6.2 is also a linguistic one and yields a collection of all sub-types of *Employee*, which include *Manager*, *Clerk*, *Receptionist* and *Consultant*. This collection is the input for the *reject* operation with the parameter `self`, which means that every element of the collection is collected and returned in a new collection except the *Manager* class. Then, this collection is input to the *forAll* collection operation, which returns *true* if the *Manager* is the highest-paid employee and false if any other *Employee* has the same or a higher salary value. If, in fact, the *forAll* operation returns false, then the invariant constraint evaluates to false and the model is, with regard to the defined constraint in Constraint 6.2, invalid.

6.2 Ontological Introspection

In the context of DOCL, the primary focus is to support constraints that navigate over lower ontological levels. This "downward" direction involves navigating from more general concepts to more specific ones, allowing the definition of constraints that should be valid for deep instances. However, the "upward" form of ontological navigation, moving from specific instances to more general concepts, can also be useful. There are scenarios where navigating upward in the ontological dimension can be beneficial. By navigating upwards, a constraint can access more abstract information about specific instances from higher-level concepts, thereby supporting queries that retrieve information from more abstract concepts and categories. This is the basis for ontological introspection in DOCL. In addition, the upward navigation direction enables, in certain cases, reasoning based on specific instances that can lead to conclusions about more general concepts or properties. Navigating upward can help in identifying broader patterns or rules that apply to a set of instances.

Once a navigation reaches a model element at a higher level, it can access all information at that level by normal intra-level navigations, such as the *creator* navigation in Constraint 6.3. The result of this kind of navigation is a set that contains every entity that can be classified by *Actor* but is connected to *TaskType* by the *createdBy* connection.

```
context Testing
inv allCreators: self.$TaskType$.creator -> size() > 0
```

CONSTRAINT 6.3: The upward ontological dimension navigation

DOCL expressions use '$' symbols to enclose inter-level ontological navigation, which like the use of the '#' symbol for linguistic navigation, leaves no room for ambiguity.

6.3 Deep Classification Operations

To check the classification relationship of model elements, OCL provides two operations, *oclIsKindOf(type)* and *oclIsTypeOf(type)*, and to retype or cast a model to another type, OCL provides the *oclAsType(type)* operation. The operations can be invoked on a source object and are then checked against the type of the passed argument. The *oclIsTypeOf(type)* operation evaluates to true, if and only if, the invoked object's type is identical to the argument. The *oclIsKindOf(type)* operation evaluates to true, if and only if, the invoked object's type is identical either to the argument or to any of the subtypes of the argument. In our terminology, therefore, *oclIsTypeOf(type)* checks if the invoked object is a direct instance of the argument, while *oclIsKindOf(type)* checks if the invoked objects are an indirect instance of the argument.

The relationship between the *O0* (Figure 3.3) and *O1* (Figure 3.4) can be used to illustrate the semantics of the *oclIsKindOf* and *oclIsTypeOf* operations. DOCL supports both of these "shallow" operations from OCL, in addition to *oclIsTypeOf*) and *oclIsKindOf* it can query *isDirectInstanceOf* and *isInstanceOf*, and also adds a new variant called *isIndirectInstanceOf* which returns true if a clabject is an instance of another clabject but not a direct instance of it. DOCL also generalizes the three new operations into a "deep" form which operates over the offspring of a clabject, not just the immediate instance at the level below. The semantics of these new operations are illustrated in 6.1. Thus, for example, while the *instanceOf* operation just checks that the supplied argument is an immediate instance of the invoked objects, at the level immediately below (e.g.) *SSDeveloper* with the name "Ann Smith" is of type *Developer*), the *isOffsrpingOf* checks whether the argument is a deep instance of the invoked objects (e.g. whether the *SSDeveloper* clabject is a deep instance of *JuniorActorType*).

The distinction between direct offspring and indirect offspring is that the set of direct offspring consists of clabjects that are reachable through the direct instances-of relationship. The indirect offspring consists of clabjects that are indirect instances and their direct and indirect instances and so on.

The same generalization is also supported in the case of the "shallow" *allInstances()* operation. When executed on a clabject, this operation returns

context SSDeveloper (Ann Smith) isInstanceOf(...)	
Developer	true
JuniorActorType	false
ActorType	false
ACMEActor	true

context SSDeveloper (Ann Smith) isOffspringOf(...)	
Developer	true
JuniorActorType	true
ActorType	true

context SSDeveloper (Ann Smith) isDirectInstanceOf(...)	
Developer	true
JuniorActorType	false
ActorType	false

context SSDeveloper (Ann Smith) isDirectOffspringOf(...)	
Developer	true
JuniorActorType	true
ActorType	false

context SSDeveloper (Ann Smith) isIndirectInstanceOf(...)	
Developer	false
JuniorActorType	false
ActorType	false

context SSDeveloper (Ann Smith) isIndirectOffspringOf(...)	
Developer	false
JuniorActorType	false
ActorType	true

TABLE 6.1: Classification checking methods [53]

the set of all instances of the element (direct and indirect). DOCL also provides a "deep" version of this operation, *allOffspring()*, that returns all offspring (direct or indirect) of the invoked clabject, over multiple classification levels.

It is also possible to define these operations as bodies of defined methods in the *Clabject* context. Constraints 6.4 to 6.9 show the defined semantics of each operation as a body constraint.

Constraint 6.4 queries for the supertypes of a clabject and adds itself to the resulting collection. In that collection, there must be a clabject for which it holds that the clabject *c* is a direct type of the clabject in question.

```
context Clabject::isInstanceOf(c:Clabject):Boolean
body: self.#getSuperTypes()# -> including(self) ->
    exists(s|s.#getDirectType()# = c)
```

CONSTRAINT 6.4: isInstanceOf operation semantics

Constraint 6.5 checks that the *getDirectType* operation returns the clabject c, only then the clabject in question is a direct instance of c.

```
context Clabject::isDirectInstanceOf(c:Clabject):Boolean
    body: self.#getDirectType()# = c
```

CONSTRAINT 6.5: isDirectInstanceOf operation semantics

Constraint 6.6 is very similar to the Constraint 6.4 but the clabject does not add itself to the collection of supertypes. It must hold that one of the supertypes has a direct type relationship to the clabject c.

```
context Clabject::isIndirectInstanceOf(c:Clabject):Boolean
    body: self.#getSuperTypes()# -> exists(s|s.#getDirectType()# = c)
```

CONSTRAINT 6.6: isIndirectInstanceOf operation semantics

Constraint 6.7 uses the *nonReflexiveClosure* operation which does not add the starting collection to the resulting collection. If the clabject in question is an offspring of c it must hold that c is somewhere in the classification chain of itself or any of the supertypes of itself.

```
context Clabject::isOffspringOf(c:Clabject):Boolean
    body: self.#getSuperTypes()# -> including(self) ->
    nonReflexiveClosure(s|s.#getDirectType()#) -> includes(c))
```

CONSTRAINT 6.7: isOffspringOf operation semantics

Constraint 6.8 uses the *nonReflexiveClosure* operation as well to determine if c is somewhere in the classification chain.

```
context Clabject::isDirectOffspringOf(c:Clabject):Boolean
    body: self -> nonReflexiveClosure(s|s.#getDirectType()#) -> includes(c))
```

CONSTRAINT 6.8: isDirectOffspringOf operation semantics

Constraint 6.9 only checks the supertypes of the clabject in question and follows their classification chain to determine if c is in the resulting collection.

```
context Clabject::isIndirectOffspringOf(c:Clabject):Boolean
    body: self.#getSuperTypes()# -> nonReflexiveClosure(s|s.#getDirectType()#)
    -> includes(c))
```

CONSTRAINT 6.9: isIndirectOffspringOf operation semantics

6.4 Level-Aware Expressions

In standard OCL, the constraints are (usually) defined on classes and evaluated on their direct instances. There is no need to specify any level ranges for evaluating the constraint. In the MLM paradigm, however, it is necessary to have some kind of designation for which level, or over many instantiation steps, the constraints should be evaluated. After the context definition of a constraint, DOCL therefore allows the level range over which the constraint should hold to be specified. The level range has to be contiguous and can not jump over a level (i.e., leave one level out) or start at a higher level than that containing the clabject that is the context of the constraint. The default, if no explicit level range is defined, is the level immediately below the context clabject. Just like in OCL where the defined constraint is meant to be executed on instances of that element. It is also possible that the definition context (i.e., the clabject or level the constraint is defined in) and the execution context (the level where the constraint has to hold) are the same.

Constraint 6.10 shows examples of the use of this range definition feature, illustrating how the range specification appears within parentheses after the context definition. The level scoping definition "(0,0)" means that the constraint is only evaluated at the level with the index 0 which is the highest level of abstraction, i.e., the highest ontological level. The combination "(0,_)" means that the constraint is valid at the level with the index 0 and all levels beneath that level (i.e., this constraint has to hold at every level). The combination "(2,2)" means that although the context of the constraint might be defined at level 0, it only needs to hold (and thus be evaluated) on model elements at the level with the index 2.

```
context JuniorActorType (0,0)
inv: ...
context JuniorActorType (0,_)
inv: ...
context JuniorActorType (0,1)
inv: ...
context JuniorActorType (0,2)
inv: ...
context JuniorActorType (1,1)
inv: ...
context JuniorActorType (1,2)
inv: ...
context JuniorActorType (2,2)
inv: ...
```

CONSTRAINT 6.10: Examples of the level-scoping expression

DOCL does not allow a constraint to define a range that is higher than the level of its context clabjects. If a model element resides at the level with the index 1, the constraint cannot be evaluated for elements at level 0. The constraint can start only at the level 1.

6.5 Deprecated Features

There are four features in OCL that are not supported in DOCL. This section elaborates on these features and explains why they have not been included.

Definition constraints: As explained in Chapter 2, *definition* constraints allow new attributes and methods to be added to UML models and their properties to be defined. In the case of attributes, their type and default value can be defined, and in the case of methods, their body can be defined. However, this capability is superfluous since these features can obviously be added in the usual way by a modeling tool, and their default values or bodies can be defined by *derive* and *body* constraints respectively. In a MLM modeling context, *definition* constraints would be problematic because they interfere with the goal of having a declarative language since the distinction between elements in a model (i.e., classes, with their attributes and methods) and in an instance of the model (i.e., objects with their slots and methods) is blurred.

This means definition constraints would either have to be repeatedly evaluated (or compiled) every time any other constraint is checked, or the tool would have to check constantly for name definitions of those constrained types.

oclIsNew() Operation: The *oclIsNew()* operation is used in *body* or *post-condition* constraints to check whether an object came into existence during the execution of the method they describe. If an object did come into existence during the method's execution, *oclIsNew()* returns True, and if it existed before the methods started executing, it returns False. The ability to detect such information relies on the language having access to a concrete, running implementation of the containing class (i.e., on the language being an action language) which is beyond the scope of DOCL. Since, by design, DOCL is a declaration language, it does not have the power of creation and thus cannot determine when objects were created.

oclInState() Operation: The *oclInState()* operation can be used in OCL expressions to check whether an instance of a class, whose run-time behavior is modeled in a UML state diagram, is in a particular abstract state defined in that diagram. The ability to detect such information relies on the language having access to even more elaborate run-time information about an executing system since it involves the detection of an abstract state that may not be encoded in normal implementation code (e.g., whether a constellation of attribute values of the object corresponds to a particular abstract state). While it is theoretically possible to arrange for a declarative language like DOCL to have access to such run-time information, and thereby to implement *oclInState()*, and still remain declarative, doing so requires a run-time implementation equivalent to that needed for an action language. It is therefore out of scope for DOCL.

Messages: The OCL has several features facilitating the description of message exchanges between executing objects. These are particularly useful for describing interaction protocols, for example. However, since their implementation would also require full access to the state of a running system, like the implementation of *oclInState()*, they are also out of scope for DOCL.

Chapter 7

DOCL Prototype Implementation

This chapter presents the prototype implementation of DOCL developed as part of this dissertation. As explained in Chapter 2, DOCL is a key part of the Melanee MLM ecosystem. The chapter therefore explains how DOCL is integrated via the tool's plugin architecture and how all the parts work together to deliver DOCL functionality.

7.1 Architecture

Figure 7.1 shows the overall architecture of Melanee, which is built on the Eclipse platform. The graphical editor of Melanee is based on the Graphical Modeling Framework (GMF) and all Melanee models, including the PLM, are represented using the Eclipse Modeling Framework (EMF). These EMF-based models are then transformed into Java code that is enhanced with OCL expressions. The model well-formedness validation is performed using a language from the Epsilon language family [76] called the Epsilon Validation Language (EVL).

The Epsilon language family provides a model management infrastructure that allows interaction with all kinds of models [76]. More specifically, it provides an interface through which languages that are not based on the MOF can be connected to the languages in the Epsilon family. The authors claim that, collectively, the Epsilon language family provides more powerful model management capabilities than OCL, such as the ability to express intra-model constraints. OCL cannot serve as an Epsilon model management language because it cannot create, update, or delete model elements or update attribute values.

In addition to Melanee's core functionality, there are multiple plug-ins that can display and edit models in a plethora of different notations, such as the *Diagram*, *Textual*, *Table* and *Form* plugins. The *Designation* plugin handles the proper designation of clabjects and prepares them for use across multiple

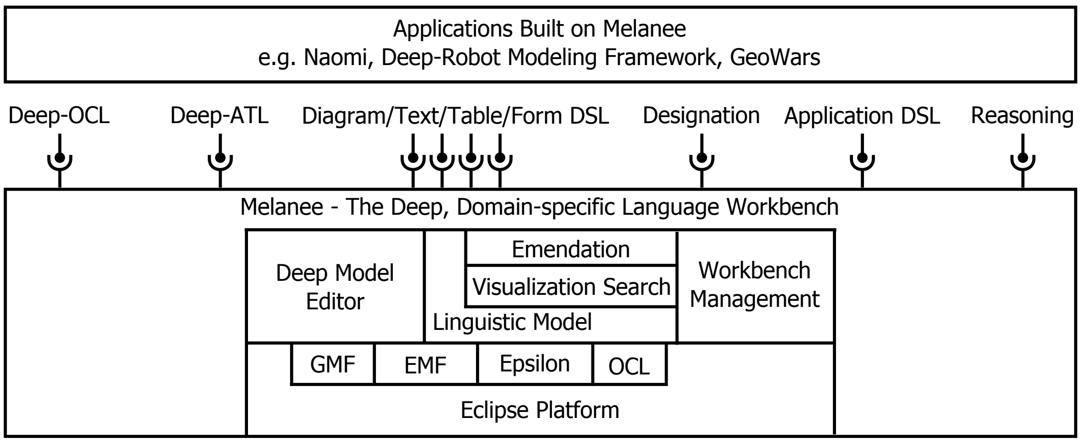


FIGURE 7.1: The Melanee Architecture [53]

models and platforms [13]. There is also the DOCL plug-in, which is the subject of this thesis. The plug-ins are hooked into Melanee via extension point interfaces that are defined in the Melanee core infrastructure. These can be defined in such a way that each plug-in can appear anywhere in the Eclipse application.

```
Data: element, constraintType
Result: constraintList
types ← element;
classification : while (type ← types.poll()) $\neq$ null) do
    types ← types ∪ type.getDirectTypes();
    superTypes ← types ∪ type.getDirectSuperTypes();
    inheritance : while (clabject ← supertypes.poll() $\neq$ null) do
        constraintList.add(getConstraintFromElement(clabject,
          constraintType);
    end
end
```

Algorithm 1: The constraint search algorithm [53]

The operation shown in Algorithm 1, inspired by the visualizer search algorithm of Gerbig [53], shows how the constraints are found for each element of the model. It has two arguments – an element of the model (i.e., a connection or an entity) and a constraint type defining the kind of constraint currently being sought, (i.e., invariant, derive, or other types of constraints). The algorithm searches up the classification hierarchy and subordinately, up each inheritance hierarchy, for constraints of that type, and puts them into the constraint list returned at the end of the execution.

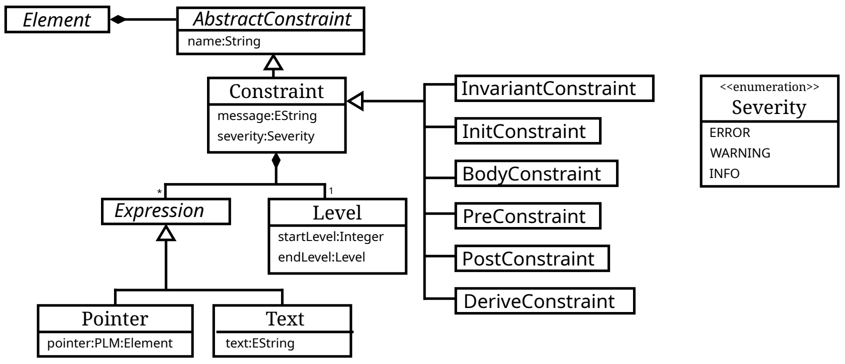


FIGURE 7.2: Meta-Model for constraints in LML

7.2 Meta-Model Definition

Figure 7.2 shows the meta-model of the constraint data structures introduced into Melanee by the DOCL plug-in. These data structures essentially constitute an extension to the PLM since they are related to the *Element* class at the root of the PLM metamodel. This allows DOCL constraints to be automatically persisted as part of an instance of a Melanee PLM model, unlike previous deep constraint languages implemented in Melanee such as [71] which did not allow constraints to be saved within deep models.

As can be seen in Figure 7.2 instances of the class *AbstractConstraint* are contained by the PLM class *Element*, which is therefore common to both meta-models. The fact that *AbstractConstraint* instances are contained in the *Element* instances means that every concrete subclass of *Element* can be enhanced with constraints, e.g., *DeepModel*, *Level*, *Inheritance*, *Connection*, *Entity*, *Attribute* and *Method*.

The *Constraint* class has six subclasses corresponding to each kind of constraint supported by the OCL. It also has two attributes, *message*, which allows users to define a message to be displayed if a constraint fails, and *severity* which allows users to define the importance of a failure to adhere to a constraint in terms of three values of an enumeration type – *ERROR*, *WARNING* and *INFO*. Each severity level is displayed with a different symbol if a constraint is violated. If a constraint is not violated, the severity attribute has no semantic effect on the model.

A *constraint* contains instances of two other classes, i.e., *Level* and *Expression*. The former encodes the execution level of the constraint (i.e., at which levels or level range the constraint should be evaluated), while the latter, an abstract class, defines how the contents of a constraint are stored. There are two options, which are concrete subclasses of *Expression*, *Pointer* which stores

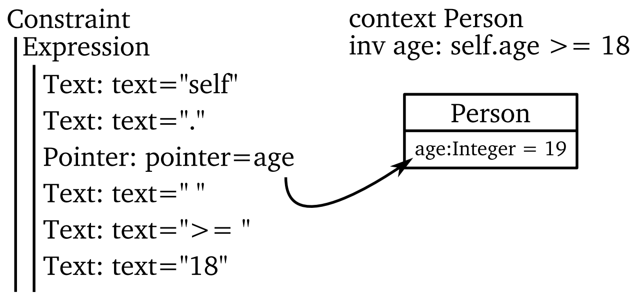


FIGURE 7.3: Semantics for saving constraints in LML

a reference to an *Element* instance in the model, and *Text* which contains a string that is not a reference to anything in the model.

7.2.1 Saving DOCL Expressions

Figure 7.3 displays an example of a DOCL expression that constrains the class *Person*. The constraint restricts the *age* attribute to be greater or equal to 18. On the left-hand side, the expression is split into the two aforementioned *Expression* classes, *Pointer* and *Text*. The saving algorithm scans the whole statement from left to right and tries to find an element that it can point to in the current navigation context. In this case, the *age* attribute of *Person* is an element the algorithm can point to. The advantage of storing it as an instance of *Pointer* instead of an instance of *Text* is that any change in the name of that element in the model is automatically taken into account when the expression is redisplayed. This saving mechanism avoids the problem of co-evolution and saves time refactoring the model or constraint expressions.

7.3 Grammar

This section gives a detailed description of how the DOCL grammar is implemented using ANTLR (version 4), and in particular how the grammar rules are defined so that the lexing process can create appropriate abstract syntax trees.

```
context Customer
inv ofAge: age >= 18
```

CONSTRAINT 7.1: Simple OCL expression

Constraint 7.1 shows a fairly simple constraint. The context is the class *Customer* and the constraint states that the *age* attribute has to have a value that is greater than, or equal to, 18. Figure 7.4 shows this constraint in the form of an abstract syntax tree. The root of the tree is the *contextDeclCS* rule,

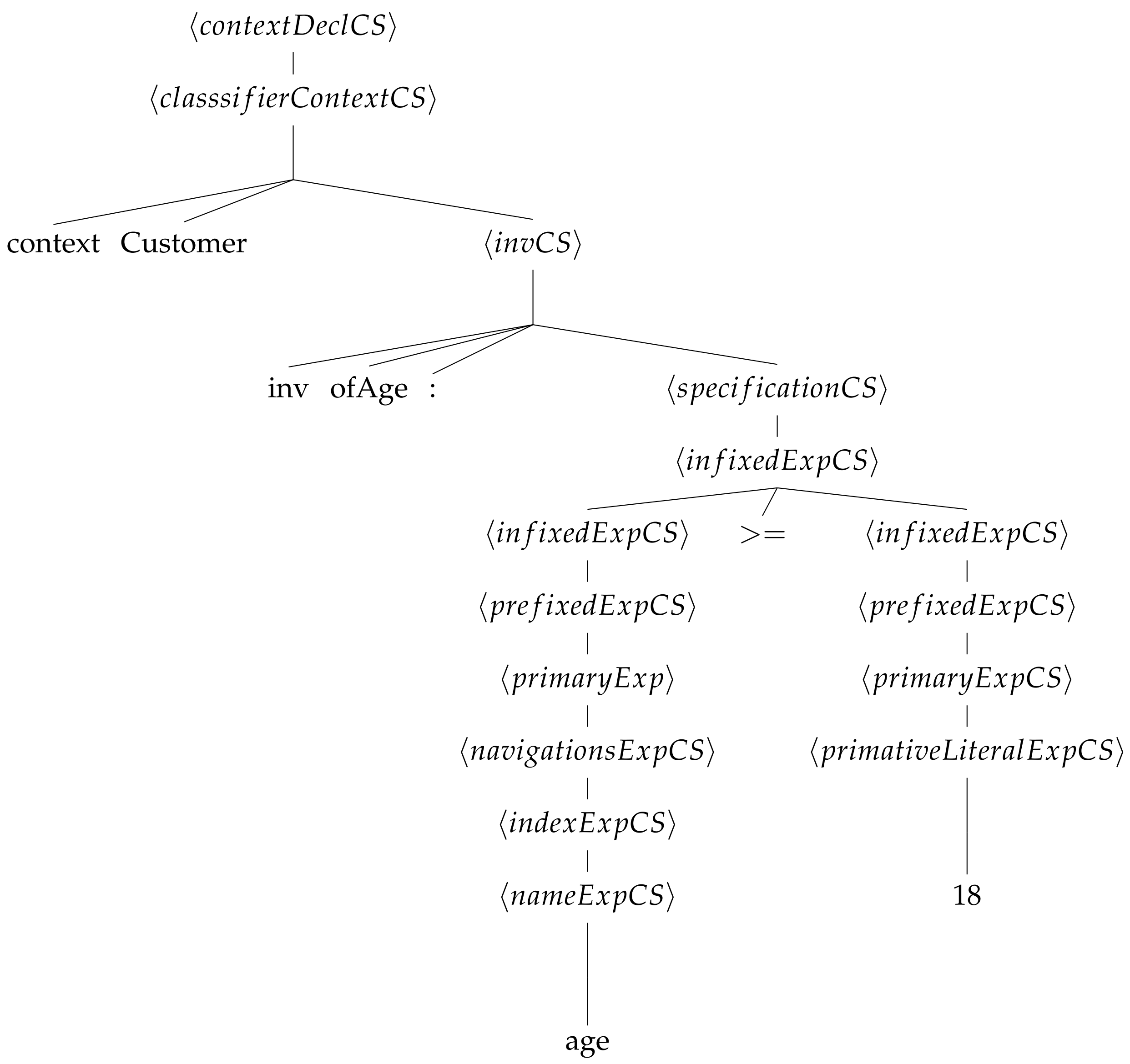


FIGURE 7.4: An example of a parsed OCL statement

which takes the context definition expression as input. The tree then fans out to other reachable rules, such as the *invCS* rule, which matches the expression "inv" (i.e., an invariant constraint), and the *specificationCS* rule, which captures the actual meaning of the whole OCL statement – namely, that the *age* attribute value has to be greater than or equal to 18. This shows how DOCL statements are processed by the parser generated by ANTLR.

Constraint 7.2 shows an OCL expression which is, in essence, a logic statement comparing four Boolean attribute values *a*, *b*, *c* and *d*. This shows the importance of grammar rule ordering to the accurate evaluation of DOCL expressions. Constraint 7.3 shows the grammar rule that is responsible for the correct ordering of the logical operators that can be used in DOCL.

```
context Person
inv logic: a and b implies c and d
```

CONSTRAINT 7.2: DOCL logic operator expression

Figure 7.5 shows the parsed statement in the form of an abstract syntax tree. According to the grammar rule *infixedExpCS*, which can be seen in Listing 7.3, the *implies* rule is at the bottom compared to the *and*, *or* or *xor* rules. That means that the *implies* block is less binding to the surroundings than the *and* blocks. It is similar to the arithmetic rules for ordering operations, like addition/subtraction and multiplication/division where the multiplication takes place before the addition. The same approach is applied here – the first *and* block containing *a* and *b* is evaluated first before the *implies* block combines the results of the first and second *and* blocks. In other words, the higher up a matching rule is in the grammar rule (i.e., the earlier it can be matched), the more closely bound it is to the surrounding expressions. Since the division and multiplication rules are also defined higher than the addition and subtraction rules, the operation ordering rules hold by default without having to check the statement twice.

```
infixedExpCS
  :
   prefixedExpCS
   | iteratorBarExpCS
   | left = infixedExpCS op =
   (
     '/'
     | '*'
   ) right = infixedExpCS
   | left = infixedExpCS op =
   (
     '+'
     | '-'
   ) right = infixedExpCS
   | left = infixedExpCS op =
   (
     '<='
     | '>='
     | '<>'
     | '<'
     | '>'
     | '='
   ) right = infixedExpCS
   | left = infixedExpCS op = '^' right = infixedExpCS
   | left = infixedExpCS op =
   (
     'and'
     | 'or'
     | 'xor'
   ) right = infixedExpCS
   | left = infixedExpCS op = 'implies' right = infixedExpCS
  ;
```

LISTING 7.3: The infixedExpCS rule as defined in the ANTLR grammar

7.4 Interpreter

The process of parsing DOCL expressions begins with the lexing of the editor input. This happens, for example, when the user requests the validation of invariant constraints or when something in the model changes to trigger the evaluation of all *derive* constraints in the model. After the lexing process has produced a token stream, the parser creates a parse tree that can be "walked over" in order to interpret the expression. The *RuleVisitor* walks the parsed tree of tokens and navigates the model with the help of the *ClabjectWrapper* class. The *RuleVisitor* class also evaluates and interprets other parts of the expression, like collection operations. Figure 7.6 displays the flow of data in the DOCL application. The result of the expression is then returned to

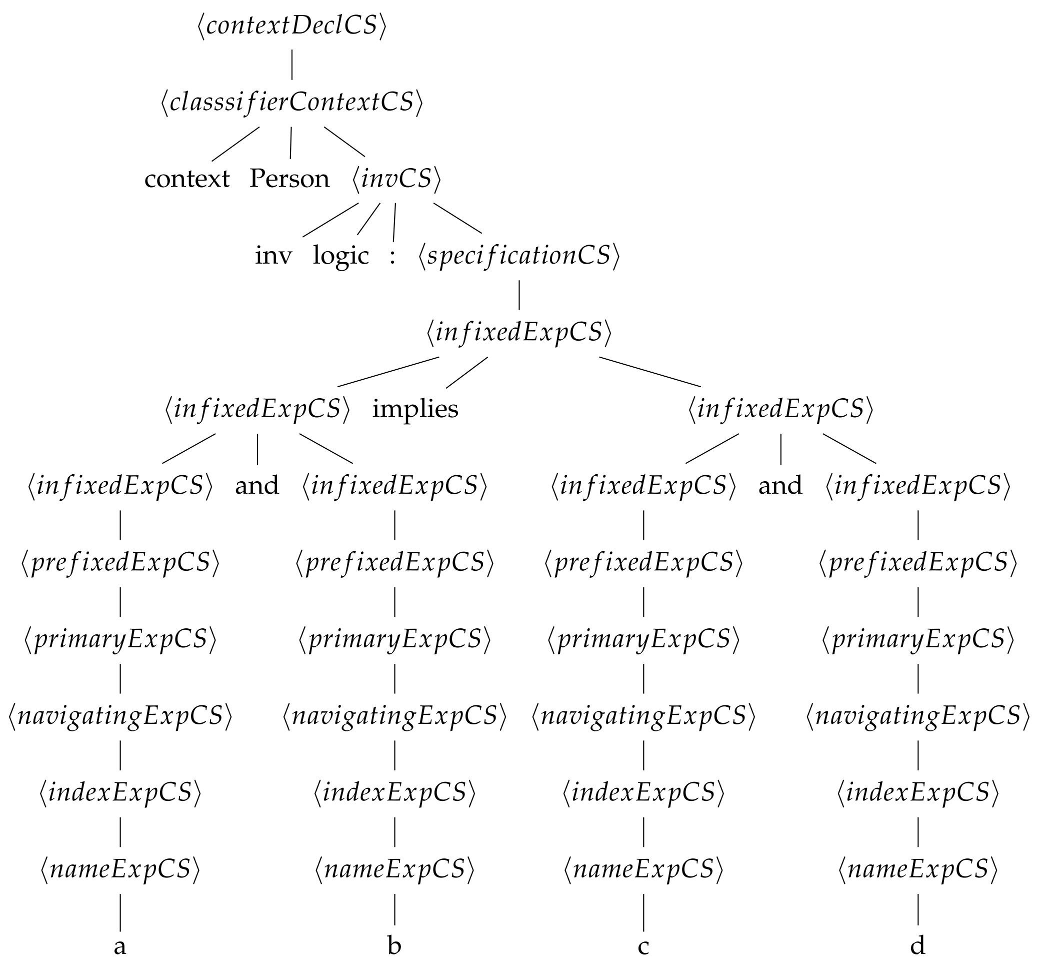


FIGURE 7.5: Example of a parse tree with a combined logical expression

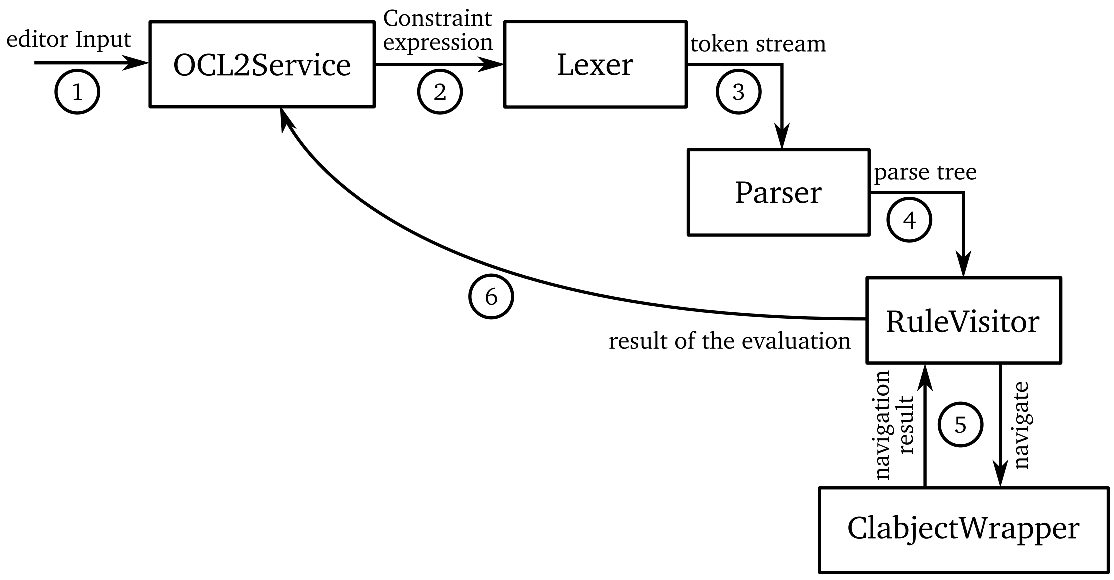


FIGURE 7.6: Data flow when executing/evaluating a constraint in DOCL

the *OCL2Service* which in turn either alters the value of derived attributes or annotates failed invariant constraints in the appropriate clabjects.

7.5 Triggering Constraint Evaluation

There are six different constraint types in DOCL – invariant, derive, init, body, pre, and post constraints – each type has a different scope with regard to the evaluation context. This section explains how, and when, each type is evaluated.

Invariant Constraint: Invariants are Boolean expressions that can be defined for *Clabjects* in the ontological dimension, not for attributes or methods, and should be evaluated to true for all instances that exist in the level range specification. It is possible to define an arbitrary number of invariant constraints on one *Clabject*. This type of constraint is not evaluated every time the model changes. The user has to proactively trigger the evaluation of invariant constraints by pressing a button.

If an invariant constraint evaluates to false, the respective *Clabject* is marked and the user is notified about the constraint violation. It is also possible to define invariant constraints on every class in the linguistic meta-model (i.e., the PLM).

Init Constraint: The init constraint can only be defined on *Attributes* of *Clabjects*. When a new instance of a *Clabject* with an associated init constraint is created, the DOCL plug is notified by the underlying framework to evaluate the defined constraint and initialize the attribute with the result of the computation. The execution is implicitly triggered when a user creates an instance of a clabject with an *init* constraint. It is only possible to define one init constraint on one *Attribute*.

Pre/Post Constraint: Both pre- and post-constraints are only applicable to methods of a *Clabject*. Preconditions check the state of a model before a method is executed, and if they are evaluated to false, the execution of the method is canceled. Postconditions check the state of a model after a method has been executed (assuming the precondition is true), and if they are evaluated as false a flag is raised. Both types of constraint are triggered automatically when a user-defined method of a model element is part of the execution of another constraint. An arbitrary number of preconditions and postconditions can be defined on a given *Method*.

Derive Constraint: Derive constraints are only applicable to *Attributes*, and a given *Attribute* can only have one such constraint. The evaluation of *derive* constraints is triggered whenever a model is changed in any way. Even changes totally unrelated to a *derive* constraint's computation will trigger its (re)evaluation. When the plug-in is notified about a change to a model element, the whole deep model is searched for this constraint type.

Body Constraint: The last constraint type defines the body of a *Method*, so there can only be one such constraint per *Method*. This constraint type is also evaluated when the operation is called as part of the execution of another constraint.

In summary, all but one type of DOCL constraint is implicitly evaluated – invariant constraints. The execution of invariant constraints has to be explicitly requested by the user via the Melanee user interface. For all other types of constraints, execution is triggered by some other modeling event. When a new *Clabject* is instantiated, the derive and init constraints are triggered to compute their respective results. Pre-, post-, and body constraints are triggered on a method when it is invoked as part of the evaluation of another DOCL expression.

Part IV

Use Cases

The fourth part of the thesis presents the main use case the DOCL language and platform were developed to support. The first chapter defines the basic notion of modeling styles that can be applied as needed, presents the core style and the default style which provide the foundation for deep modeling using the Melanee tool, and shows how DOCL supports their definition in a highly precise, concise and easy-to-understand way. The second chapter presents a series of optional modeling styles that allow modelers to control the well-formedness of their models at the granularity of the whole level and deep model. In contrast, the third chapter presents a series of modeling patterns that apply at a smaller level of granularity to specific groups of model elements.

Chapter 8

Deep Modeling Styles

Visual models of the kind popularized by the UML are constructed using a long-established set of conventions and rules, many of which arise from the classic, two-level way of categorizing objects. For example, when classifying the objects that can exist in a universe of discourse it is usually assumed that all objects in that universe of discourse are classified, and when classifying the elements that can make up a model (i.e., when meta-modeling) it is usually assumed that all such elements need to be classified. Similarly, when a stack of model levels is created through meta-modeling, it is usually assumed that classified model elements occupy the level below the elements that classify them.

Because these conventions seem so natural and are ingrained into how modelers think, they are usually left implicit. Indeed, when meta-models are used to define the linguistic concepts needed in a modeling environment (i.e., to define the language used for modeling), as in MOF-based environments, there is no option but to classify all the model elements that can be used at the level below since model elements cannot be created other than by instantiating their type. However, for modeling environments based on the OCA, this assumption, and many other fundamental conventions of modeling, no longer automatically apply in the ontological dimension. This is because, in the OCA, the basic existence of clabjects in the ontological modeling space is enabled by their linguistic type, so there is no longer an inexorable need for them to have a direct ontological type, and if they have one, for it to be at a particular level.

Although the linguistic dimension of the OCA is still constrained by the traditional conventions of classic, two-level modeling, therefore, this is no longer the case for the ontological dimension. The underlying rules governing the relationships between model elements in the ontological dimension (including classification) can therefore be optimized for different modeling goals. Moreover, since the ontological dimension is fully described by the linguistic metamodel, including the basic notions of “deep model” and “level”, a level-aware constraint language like DOCL can be used to influence and enforce various sets of modeling rules on one, or more, of the ontological levels - that is, can be used to enforce deep modeling “styles”. This section provides a more concrete definition of the notion of “style” in deep modeling and shows how DOCL can be used to define and enforce a selection of common modeling styles.

8.1 What is a Modeling Style?

The notion of modeling styles has not been widely used in the modeling community because the fundamental rules of classic, two-level modeling cannot be circumvented when tool support is required. Regardless of how reflective a language definition may be, a tool needs to know what fundamental, inbuilt model elements have to be supported. The notion of modeling styles has not yet gained acceptance in the MLM community because the flexibility offered by the OCA is still being researched. In fact, the differences between the many MLM tools and languages available today can often be understood as differences in styles rather than in basic concepts, such as clabjects and potency, which have become widely accepted. Rather than hard-wiring these alternative styles in different fixed languages and tools, deep reflective constraint languages like DOCL make it possible for an OCA-based tool to support multiple styles in a flexible and interchangeable way.

To this end, we define a deep modeling style in the context of the OCA as follows –

Definition 8.1 (Modeling Styles)**.** A deep modeling style is a coherent ensemble of rules that govern the structure and/or relationships between all the model elements in at least one, and usually more, ontological levels of a deep model. The coherent ensemble of rules that characterize a style is referred to as a style definition.

To be regarded as a “style”, therefore, an ensemble of rules must apply uniformly and fully to all model elements in at least one ontological level, without exception. However, often the rules apply to most, if not all, ontological levels in a deep model. The one common exception is the top (most abstract) level, which is often exempt from the rules that apply to the other models because it plays a special, foundational role.

Note that in principle, all the rules that govern MLM in an OCA modeling environment can be defined within style definitions, since at its core the linguistic metamodel contains nothing more than the syntax of the model elements that exist in the ontological dimension. However, in practice, it is likely that OCA-based tools will define a “core style” that has to be adopted and cannot be changed. The following subsections present some important styles that are defined using DOCL constraints.

8.2 The Core Style

This section presents a core modeling style that defines the minimal, core set of rules common to the majority of all deep modeling approaches. All other styles can therefore be said to “subsume” the core style, and essentially extend it with additional, stronger constraints.

8.2.1 Orderly Hierarchies

Many of the differences between MLM revolve around the properties of three core hierarchies and how they can be related in models – ontological levels, classification (i.e., instanceOf relationships), and inheritance (i.e., sub/super-type relationships). This section identifies three underlying properties that these hierarchies must have in all MLM approaches.

Absence of Non-monotonic Abstraction: One of the most fundamental aspects of any multi-level modeling approach is how classification relationships relate to the definition of the levels. As mentioned, in Chapter 3, Melanee enforces strict modeling, but this is not embedded in the PLM itself. In fact, the PLM does not directly relate the notion of abstraction, bound up in the relationship between a type and its instances, with a stack of ontological levels whose ordering is characterized by their labels (i.e., *O0*, *O1*, *O2*, ...). However, all MLM languages that use the type/instance relationship as the basis for defining levels, also agree on a fundamental relationship between them which here is called the principle of “Monotonic Abstraction”.

As shown in Figure 8.1, this is the basic idea that for all clabjects, the ontological types of those clabjects (which by definition are more abstract), must reside at a “higher” level. In other words, the types of clabjects cannot occupy higher levels than those clabjects in some parts of a multi-level model and lower levels in other parts. Such a structure can be thought of as

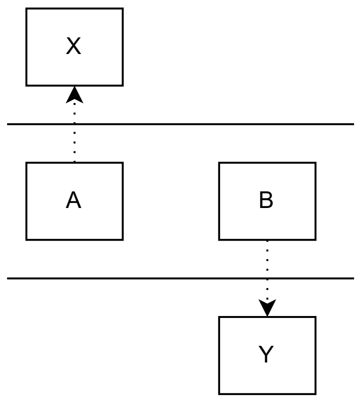


FIGURE 8.1: Non-Monatonic abstraction anti-pattern

an "anti-pattern" so that the corresponding style is essentially the rule that this Non-monotonic Abstraction anti-pattern should be absent from (i.e., not occur within) a deep model.

Constraint 8.1 shows a DOCL constraint that captures this basic rule governing the relationship between classification and levels in deep modeling.

```
context Clabject(0,_)
inv: self.doclGetDirectInstances() ->
    forAll(inst|inst.#getLevelIndex()# > self.#getLevelIndex()#)
```

CONSTRAINT 8.1: Absence of non-monotonic abstraction style

Absence of Circular Classification: A fundamental property that all classification hierarchies should adhere to, regardless of their relationships to levels, is that there should be no circular classification situations in which, for example, C is an instance of B, B is an instance of A and A is an instance of C, as illustrated in Figure 8.2. Such a circular instance of chain can also be thought of as an "anti-pattern" so that the corresponding style is essentially the rule that this Circular Classification Anti-pattern should be absent from (i.e., not occur within) a deep model.

The most concise way of defining this style is to use a variant of the OCL closure operation, which returns the transitive closure of transitive relationships between clabjects including the clabject from which the closure "search" was initiated. The DOCL constraint for defining this constraint, Constraint 8.2, used the *nonReflexiveClosure* variant which returns the same set as *nonReflexiveClosure* but without the starting clabject (i.e., the clabject referred to as *self*). Only if the element that is referenced as "self" is not in the collection returned by *nonReflexiveClosure* it is certain that there is no instance cycle (i.e., the occurrence of the anti-pattern). If the "self" element is present

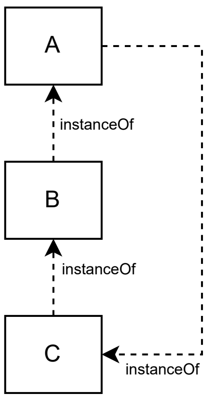


FIGURE 8.2: Circular Classification Relationship Anti-pattern

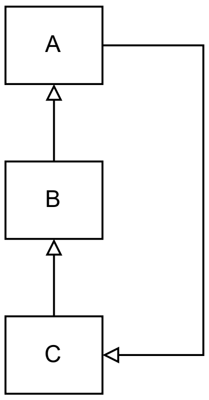


FIGURE 8.3: Circular Inheritance Relationship Anti-pattern

in the resulting collection a cycle is present and the Melanee tool will annotate the clabject with an error marker and inform the user that this particular clabject is part of an anti-pattern.

```
context Clabject(0,_)
inv: self -> nonReflexiveClosure(clabject|clabject.#getDirectTypes()#) ->
    excludes(self)
```

CONSTRAINT 8.2: Absence of circular classification style

Absence of Circular Inheritance: The same basic problem of circularity can exist in inheritance hierarchies, so another constraint is needed to prevent the Circular Inheritance Relationship Anti-pattern. As shown in Figure 8.3, this anti-pattern is very similar to the circular classification relationship but instead of the classification relationship the clabjects involved are connected via inheritance relationships. This anti-pattern can occur in MLM approaches that use the level concept since all involved clabjects can exist at the same level.

The following constraint in Constraint 8.3 can be used to detect the circular inheritance relationship anti-pattern. Instead of constraining the classification hierarchy, the constraint rules out circular chains of inheritance.

```
context Clabject(0,_)
inv: self -> nonReflexiveClosure(clabject|clabject.#getSupertypes()#)
    -> excludes(self)
```

CONSTRAINT 8.3: Absence of circular inheritance style

8.2.2 Deep Characterization

MLM approaches like LML that support deep instantiation uses so-called vitality properties to describe the relationship between types and their instances, and also to indicate what roles clabjects play (i.e., type and/or instance). The LML defines three vitality properties - potency, durability, and mutability, but most deep-instantiation-based MLM languages support fewer. For example, MetaDepth only uses potency, while DLM uses potency and durability. However, whichever selection of properties they support, certain core principles must always apply, not only related to the basic definition of classification but also to the preservation of the Liskov Substitutability Principle [87] in inheritance hierarchies. This section presents these principles for each of the three vitality properties.

Potency: The original and most important vitality property is the "potency" property, which is the basic mechanism by which deep-instantiation-based MLM languages support deep characterization. Although such languages sometimes have different rules for potency, the majority agree on a principle that is sometimes referred to as "characterization potency" [77]. This is a relaxed version of the original potency rules first introduced in [12]. In simple terms, the relaxed rules simply require the potency of a direct instance of a clabject to be lower than the potency of that clabject. The DOCL constraint showing this basic characterization rule for potency is shown in Constraint 8.4, where the potency of a clabject is a non-negative integer.

In order to preserve the Liskov Substance Principle for direct and indirect instances of a clabject, there are also several fundamental rules governing the relationship between the potencies of clabjects that are in an inheritance relationship. These are summarized in Table 8.1, where the columns represent the potency value of the super-clabject in a specialization relationship and the rows represent the potency values of the sub-clabject.

The first row indicates that a sub-clabject can have a potency of 0 regardless of the potency of its super-clabject as long as it also has at least one descendent with a potency greater than 0. However, if the super-clabject is potent (columns 2 and 3) none of the descendants of the sub-clabject can have a potency greater than the potency of the super-clabject. If the sub-class is impotent, none of the descendants of the sub-class can have a potency greater than the potency of the super-clabject's closest potent ancestor, if it has one.

```
context Clabject(0,_)
inv: self.doclGetDirectInstances() ->
    forAll(p|p.#getPotency()# < self.#getPotency()#)
```

CONSTRAINT 8.4: Characterization Potency Style: potency of direct instances must be one lower

Any clabject in an inheritance chain other than the bottom one can be impotent (i.e. an abstract class). However, if one or more of the super-clabjects of an abstract clabject is potent, to ensure that all instances of the abstract clabject are also instances of its potent super-clabjects the potencies of the abstract class's descendants must not exceed the potencies of any of its potent super-clabjects. In the two right-hand cells of the table, the abstract class in question does have a potent super-clabject, so this requirement applies. However, an abstract clabject is not always required to have potent super-clabjects. Thus, in the top left-hand cell of the first row, the requirement only applies if the abstract clabject has at least one potent super-clabject.

The second row indicates that if the potency of the sub-clabject is 1, the super-clabjects can have any potency. For the left-hand cell of this row, when the super-clabject is impotent, there are two situations. Either the super-clabject has a potent ancestor or it has not. In the first case, the minimum size of this potency must be 1, so the sub-clabject's potency does not exceed it. In the second case, there is no problem because a chain of abstract clabjects can have potent sub-clabjects of any potency. For the other two cells in this row, the potency of the sub-clabjects is guaranteed to be less than the potencies of the super-clabjects, so the corresponding situation is always possible.

In the case of the third row, the bottom-right cell of Table 8.1 defines the basic rule that applies when potent clabjects are involved in inheritance relationships – a sub-clabject's potency must be less than or equal to the super-clabjects potency. One consequence of this is that a super-class with potency 1 cannot have sub-clabjects of potency 2..*, so the middle cell of the bottom row is not possible. As usual, the case when the super-clabject is impotent is special (left-hand cell of the bottom row). Again, all sub-clabjects of the

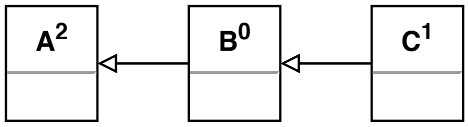


FIGURE 8.4: Inheritance example with different potencies

Sub \ Super	0	1	2..*
0	the sub-clabject must have at least one descendant with a potency >0 and no descendants with a potency greater than the potency of the super-clabject's closest potent ancestor if there is one	the sub-class must have at least one descendant with a potency >0 and no descendants with a potency greater than that of the super-clabject	the sub-clabject must have at least one descendant with a potency >0 and no descendants with a potency greater than that of the super-clabject
1	Possible	Possible	Possible
2..*	the closet potent ancestor of the super-clabject, if there is one, must have a potency at least as large as the sub-clabject's potency	Not possible	The sub-clabject's potency must be less than or equal to the potency of the super-clabject

TABLE 8.1: Inheritance Potency rules [85]

abstract clabject must also be instances of all of its potent super-clabjects. Thus, if the abstract clabject has any potent super-clabjects, the closest ancestor to it in the inheritance chain must have a potency that is greater than or equal to that of the abstract clabject's sub-clabject. This is sufficient because any potent descendants further down the inheritance hierarchy are bound to have equal or higher potencies. Similarly, it is sufficient for the closest ancestor's potency to be no greater than the sub-clabject's because any of its super-clabjects must have equal or lower potencies.

Figure 8.4 clarifies the rationale behind the top right-hand cell of Table 8.1. A clabject with potency 2, *A*, can have a sub-clabject that has potency 0,*B*, an abstract clabject, but only if *B* has a sub-clabject, *C*, which has a potency higher than 0 but less than or equal to the potency of *A* (the closet potent ancestor to *B*).

The DOCL constraint that defines these potency rules is shown in table

8.1. The *getSubtypes()* operation returns an ordered set so that the clabjects in that collection are ordered in terms of their occurrence down the inheritance chain.

```
context Clabject(0,_)
inv: if self.#getPotency()# = 0
        then (self.#getSubtypes()# -> exists(sub|sub.#getPotency()# > 0))
        else (self.#getSubtypes()# -> forAll(sub|sub.#getPotency()# <=
            self.#getPotency()#))
    endif
```

CONSTRAINT 8.5: Core Style: Potency rules of inheritance

Durability: As explained in Chapter 3, durability is an endurance property of attributes that primarily governs when instances of a clabject must possess an attribute and what that implies for its instances. The basic rule for durability is similar to that for potency – if a clabject I is an instance of another clabject T which has an attribute named A with a durability greater than 0, then I must also have an attribute named A of the same type, but with a durability that is one lower. If, however, the durability of A is 0, I need not have an attribute named A of the same type. Like potency, durability is a non-negative integer value.

The DOCL constraints that define these basic rules for durability are defined in Constraint 8.6.

```
context Clabject(0,_)
inv: if self.#getDirectType()# -> size() = 1
        then true
        else self.#getDirectType()#.#getDefinedAttributes()# ->
        select(attr|attr.#getDurability()# > 0) -> collect(#name#) ->
        includesAll(self.#getAllAttributes()# -> collect(#name#))
        endif
```

CONSTRAINT 8.6: Durabilty classification rules

Ensuring that the Liskov Substance Principle is adhered to also leads to the following fundamental rules governing the relationship between the mutabilities of attributes owned by clabjects which are in an inheritance relationship. These are summarised in Table 8.2. Importantly, this table remains completely unaffected by the potency values of the clabjects that own the attributes involved.

The columns represent the durability of an attribute of the super-clabject (i.e. the super-attribute) in an inheritance relationship while the rows represent the durability of the corresponding (i.e. inherited) attribute of the subclabject (i.e. the sub-attribute), where the label 2..* stands for all durability values from 2 onward.

Sub \ Super	0	1	2..*
0	Possible	Not possible	Not possible
1	Possible	Possible	Not possible
2..*	Possible	Not possible	durabilities must be the same

TABLE 8.2: Inheritance Durability rules [85]

The first column of table 8.2 indicates that if a sub-clabject inherits a non-durable attribute (i.e. an attribute with zero durability), that attribute can have any durability. This is because non-durable attributes have no effect on the intension of a clabject and thus place no constraints on the sub-clabjects. Indeed, a sub-clabject does not even need to inherit a non-durable attribute since the intension of the super-clabject does not require the attribute's presence in instances.

The rest of the table basically shows that if the durability of the super-attribute is non-zero the sub-attribute must have the same durability. Durability rules are easy to summarize. The durability value has to stay the same except in the case of 0. If the super-type attribute has a durability value of 0 the sub-type attribute's durability value can be any positive integer value. The following DOCL constraint defines these durability rules.

```
context Clabject(0,_)
inv: self.#getDirectSupertypes()#.#getAllAttributes()# ->
     reject(a|a.#getDurability()# = 0) ->
     forAll(a| let name:String = a.#getName()# in
     self.#getAttributeByName(name)#.#getDurability()# = a.#getDurability()#)
```

CONSTRAINT 8.7: Durability inheritance rules

Mutability: Chapter 3 also explains that mutability is another endurance property of attributes that primarily governs how the values of an attribute can vary over an instantiation chain. The basic rule for mutability is that if a clabject I, which is an instance of another clabject T, has an attribute A with the same name and type as an attribute A of T, according to the durability rules, the mutability of I's A attribute must be one less than the mutability of T's A attribute unless the mutability of T's A attribute is 0, in which case the mutability of I's A attribute is also 0. Like potency and durability, mutability is a non-negative integer value. The DOCL constraint defining these rules is shown in Constraint 8.8.

Super / Sub	0	1	2..*
0	sub-attribute value must be the same as super-attribute value	Possible (attribute values can be different)	Not possible
1	Not possible	Possible (attribute values can be different)	Not possible
2..*	Not possible	Not possible	mutability values must be the same (attribute values can be different)

TABLE 8.3: Mutability rules (Durability >0) [85]

```
context Attribute(0,_)
inv: self.#getClabject()#.#getDirectType()#.#getDefinedAttributes()# ->
    reject(a|a.#getDurability()# = 0) -> forAll(a|
    if a.#name# = self.#name# and a.#getMutability()# > 0
        then a.#getMutability()# > self.#getMutability()#
        else if a.#name# = self.#name# and a.#getMutability()# = 0
            then self.#getMutability()# = 0
            else true
        endif
    endif)
```

CONSTRAINT 8.8: Mutability classification rule

The rules governing the relationship between the mutabilities of attributes owned by clabjects that are in an inheritance relationship are summarised in Table 8.3.

As mentioned above, if the durability of the super-attribute is zero, the sub-attribute can have any properties it likes because it is not of interest to the intension of the super-clabject. If the super-attribute is durable, however, the sub-attribute has to have the same durability as the super-attribute, otherwise, mutability is irrelevant. Thus, the rules in Table 8.3 assume that the sub- and super-attributes have non-zero durabilities of the same value.

The first column shows that if the super-attribute is immutable (i.e. has mutability 0), the super-attribute must also be immutable. Moreover, it must also have the same value as the super-attribute. Column 2 shows that if the super-attribute has a mutability of 1, the sub-attribute can have a mutability of 0 or 1. This is because attributes with mutability 1 and 0 both have instances with mutability 0. However, in this case, the sub-attribute and super-attribute do not need to have the same value since the super-clabject's intension allows the value of a mutable attribute to be changed by its instances.

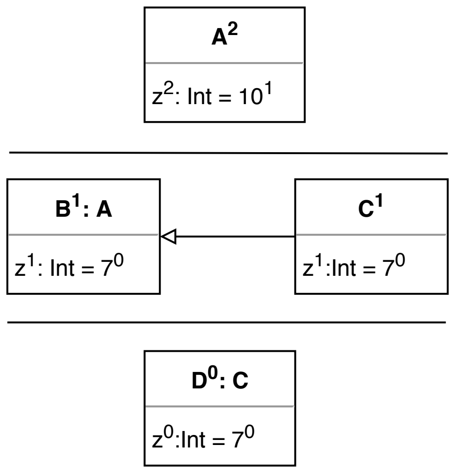


FIGURE 8.5: Inheritance example with mutability 0 values

Finally, the third column shows that if the super-attribute has a mutability greater than one, the sub-attribute has to have a mutability of the same value. However, the value of the sub-attribute can again be different from the value of the super-attribute.

Figure 8.5 clarifies the first column and first row of Table 8.3. Since the *z* attribute of the clabject *B* has mutability zero, along with non-zero durability, the *z* attribute of the clabject *C*, a sub-clabject of *B*, must have the same durability, the same mutability and the same value (i.e. 7). If the mutability of the *z* attribute of the clabject *B* were greater than 0, the *z* attribute of *C* could have a different value, although its mutability would have to be the same.

```
context Attribute(0,_)
inv: let mutability:Integer = self.#getMutability()#  in
self.#getClabject()#.#getDirectSupertypes()#.#getAllAttributes()# ->
select(a|a.#getName()# = self.#getName()# and a.#getDurability()# > 0) ->
forAll(a| if a.#getMutability()# = 1 and mutability >= 1
    then false
    else if mutability <> a.#getMutability()#
        then false
        else true
        endif
    endif)
```

CONSTRAINT 8.9: Mutability inheritance rules

8.3 The Melanee Default Style

The core style described in the previous section captures the core principles and rules that underpin any deep modeling approach using the modeling concepts provided by the LML language, as defined in the PLM. These are currently hardwired into the Melanee, which is currently the only tool that

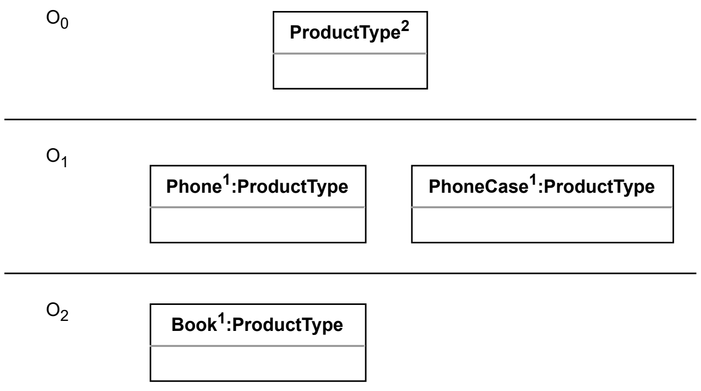


FIGURE 8.6: Example of strict multi-level modeling

supports LML-based deep modeling. However, Melanee also comes preconfigured with other modeling choices which, although not in the core, are also part of the out-of-the-box deep modeling approach users are offered when using Melanee. In other words, Melanee is preconfigured with a default style that extends the built-in rules with additional modeling rules. These are described in this section.

Strict Modeling: The core Monotonic Abstraction Levels principle described in subsection 8.2.1 only requires that an instance of a clabject reside at a lower level than that clabject, but does not stipulate which lower level. Melanee adopts a stricter style, the classic "strict (meta) modeling" style, which states that the instances of a clabject must be at the level directly below that clabject and that only classification relationships can cross level boundaries. Strict modeling is therefore compatible with the monotonic abstraction principle but strengthens it. Strict modeling therefore forms part of Melanee's default style, which subsumes the core style outlined in the previous section. Although this strict style is currently hardwired in Melanee, in the following we present the DOCL rules that define this style.

The strict deep modeling style adds two additional rules to the core style:

SR1 The only level-crossing relationship that can cross levels is the instance-of relationship

SR2 Instances of a clabject have to exist at the level exactly below that clabject

Figure 8.6 shows a few examples of what strict modeling looks like and what models are invalid in the strict modeling style. The clabject *ProductType* in the top level does not need to have an ontological type and has a potency of

2. The valid instances in this diagram are the *Phone* and *PhoneCase* clabjects, which is an instance of *ProductType*. The *Book* clabject has the correct potency value of 1 but is placed on a level too far below, and is thus not a valid clabject in the strict modeling style.

```
context Connection(0,_)
inv: self.#getParticipants()# -> forAll(p|p.#getLevelIndex()# = self.#
    getLevelIndex()#)
```

CONSTRAINT 8.10: Strict modelling style: only instance-of relationships can cross levels

These two rules can easily be defined in DOCL using the following three constraints. The Constraint 8.10 defines the context in *Connection* and ensures that every connected entity is at the same level, i.e., that the connection does not cross a level boundary. This is achieved by storing information about which level contains a connection and then comparing that with the level information of every participant of that connection.

Similarly, all clabjects in a generalization (inheritance) set have to be at the same level as well. Constraint 8.11 checks that all subtypes of a clabject must be at the same level as the supertype clabject.

Both these constraints deal with **SR1** since these types of relationships are the only things that can connect clabjects that must not cross any level boundaries.

```
context Inheritance(0,_)
inv: self.#getSupertypes#.#levelIndex# -> asSet() -> size() = 1
    and self.#getSubtypes#.#levelIndex# -> asSet() -> size() = 1
    and self.#getSupertypes#.#levelIndex# -> asSet() =
    self.#getSubtypes#.#levelIndex# -> asSet()
```

CONSTRAINT 8.11: Strict modelling style: sub- and supertypes have to be on the same level

Finally, Constraint 8.12 ensures that every clabject in a model only has instances at the level directly below. The context here is *Clabject* and the constraint is evaluated for every linguistic instance of *Clabject*. For every clabject, the index of the level occupied by its instances must be one higher than the index of the level occupied by its direct type, meaning that it occupies the level immediately below that type. This constraint deals with **SR2**.

```
context Clabject(0,_)
inv: let levelIndex:Integer = self.#getLevelIndex()# in
    if self.doclGetDirectInstances() -> size() = 0
    then true
    else self.doclGetDirectInstances() -> forAll(c|c.#getLevelIndex()# =
    levelIndex + 1)
    endif
```

CONSTRAINT 8.12: Strict modeling style: instances of a clabject have to be at the level immediately below

Chapter 9

Multi-Style Modeling

The core style described in the previous chapter is hardwired into Melanee (i.e., encoded in Java) since DOCL was not available at the time it was constructed. However, with the availability of DOCL only the basic ability to handle the PLM abstract syntax needs to be hardwired into the core. DOCL makes it possible to define and apply different modeling styles, as long as they are consistent with (i.e., subsume) the core style. This chapter presents some of the most important and useful optional modeling styles that can be defined using DOCL.

9.1 Level Organization Styles

The previous chapter presented DOCL encodings of two rules, **SR1** and **SR2**, that define and enforce the principle of strict modeling. By removing one, or both, of these rules, whilst still retaining the core principle of monotonic abstraction, less strict deep modeling styles can be applied. These give modelers more flexibility when modeling but also increase the risk that malformed models will be created.

Level-Leaping Deep Modeling: One less strict variant is to keep **SR1** but drop **SR2**. This falls back on the monotonic abstraction rule that an instance of a clabject must occupy a lower level than that clabject, but not necessarily the level immediately below, whilst retaining the rule that only classification relationships can cross levels.

This style can be useful in some domains to reduce the number of redundant clabjects at intermediate levels. It is a style supported by the Metadepth deep modeling tool [41] where it is implemented by the "leap potency" mechanism.

Level-Crossing Deep Modeling: Another less strict variant is to keep **SR2** but drop **SR1**. This retains the strict principle that instances of a clabject must occupy the level immediately below that clabject, but allows other kinds of relationships, beyond just the classification relationship, to cross level boundaries.

This style can help reduce the number of redundant clabjects when an abstraction, as well as instances of that abstraction, needs to have connections to a given clabject. A commonly used example is when *Jony Ive*, the famous "designerOf" the *iPhone*, is also the "ownerOf" an *iPhone* instance. The strict deep modeling style requires redundant representation of the same domain entity to handle such cases [83]. The level-crossing deep modeling style is supported by MLT [29] and DeepTelos [70].

Relaxed Deep Modeling: The most "relaxed" variant is to drop both **SR1** and **SR2** and retain only the core monotonic abstraction rule for deep modeling. This still enforces a clear notion of levels, including restrictions on the relative levels clabjects can occupy based on their classification relationships, but allows some freedom as to the exact levels that clabjects occupy. It also places no restrictions on what kinds of relationships can cross levels.

This style, which has been referred to as "loose multi-level modeling" in previous papers [20], supports the most concise models of a domain but increases the risk that modelers will create ill-formed models. Approaches that support this style include FMML^x [36], and DLMA [117].

9.2 Classification Styles

The previous section discussed styles dealing with the relative location of types and their instances within a stack of ontological levels, as well as the nature of the relationships that can cross levels. However, these styles do not address the issues that arise when classification relationships can, or are required to, exist between clabjects and what such relationships entail when they do exist. Note that the existence of a classification relationship between a clabject I and a clabject T means that I is a direct instance of T and T is a direct type of I. This section discusses several deep modeling styles that address different aspects of these questions.

9.2.1 Ontological Classification Mandation

The first set of classification styles deals with the question of whether clabjects should, or must, have ontological types (i.e., with classification mandation). The original papers on strict modeling strongly implied, but did not explicitly state, that all clabjects in a deep model, except that at the top (most abstract) level, must have direct ontological types. In other words, ontological classification was mandatory at all but the top level.

In traditional constructive modeling environments where instances are always created from types, this mandatory ontological classification approach for all objects is unavoidable because objects are created according to the template defined by their types. However, in OCA-based tools, ontological classification is not essential because objects are existentially created from their linguistic type. The idea that there might be a practical benefit of allowing clabjects in a model to not have an ontological direct type was first suggested by De Lara, Guerra, and Cuadrado [43], under the term "linguistic extensions". There are three fundamental deep modeling styles providing different approaches to the question of when clabjects must have an ontological classifier (i.e., type).

Universal Ontological Classification: The style implied by the earliest languages and tools for deep modeling can be characterized as "Universal Ontological Classification". This basically makes it mandatory for all clabjects in a deep model, except the clabjects at the top level, to have a direct ontological type. More specifically, unless a clabject occupies the top level, it must have one and only one direct ontological type, but can of course have multiple indirect ontological types through inheritance.

```
context DeepModel(0,_)
inv: Clabject -> reject(c|c.#getLevelIndex()# = 0) ->
    forAll(c|c.#getDirectType()# -> size() = 1)
```

CONSTRAINT 9.1: Universal Ontological Classification Style

The DOCL constraint defining this style is Constraint 9.1. This style is useful when using an ontological level to define a DSL since the goal in such a use case is to make sure that only concepts in (the abstract syntax of) the language are used in the level, or levels, below.

Selective Classification Mandation: For some applications, the universal ontological classification style is too restrictive. When using the powertype pattern, for example, the universal classification style can make it difficult to apply this pattern in a clean and simple way. Often, it is sufficient to mandate that only one of the two kinds of clabjects supported in the LML (i.e., Entities or Connections) must have a direct ontological type. This is therefore referred to as "Selective Classification Mandation".

The DOCL Constraint 9.2 defines a modeling style where the mandation of ontological classification is restricted to connections. In other words, it requires all connection clabjects, except those occupying the top level, to have a direct ontological type, but allows entity clabjects to be ontologically untyped. Of course, the opposite choice is also possible where only entity clabjects are mandated to have a direct ontological type.

```
context DeepModel(0,_)
inv: Connection -> reject(c|c.#getLevelIndex()# = 0) ->
    forAll(c|c.#getDirectType()# -> size() = 1)
```

CONSTRAINT 9.2: Mandatory Classification for Connections Style

This mandatory connection classification style is useful in deep models that include powertypes because, as mentioned above, it is useful to allow some of the participating entity clabjects to be ontologically untyped, but not at the expense of allowing all connections to be ontologically untyped. The mandatory connection classification style therefore allows the former but forbids the latter.

9.2.2 Instantiation Form Mandation

The previous set of styles addressed the question of when clabjects must have a direct ontological type but did not deal with the question of how fully a direct type must characterize its direct instances. This relates to the issue of whether a direct instance of a clabject must be an "isonym" of that clabject or might be a "hyponym" of it. In both cases, the clabject must have the features (i.e., attributes, methods, and mandatory connections) required by the intension of its direct types, but in the former case, it has only those features, while in the latter case, it has more.

Figure 9.1 shows an example of both isonynymic [72] and hyponynic instantiation. The *Phone* clabject is an isonym of *ProductType* because it has the features required by the intension of *ProductType*, but no more. The clabject *Book*, however, has an additional feature, the attribute *numberOfPages*, which is not part of the intension of the type *ProductType*. Therefore, *Book* is a hyponym of *ProductType*. Note that the isonym/hyponym distinction only

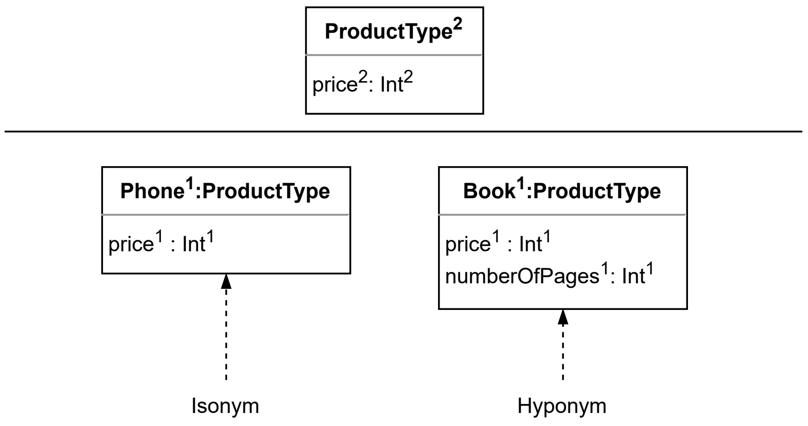


FIGURE 9.1: Example for isonynimic and hypominic instantiation

considers the potent features of a clabject's direct type (i.e., features with a potency greater than 0). This is because the impotent features (i.e., with potency = 0) of a clabject are not part of its intension and therefore do not affect its instances.

The isonym/hyponym distinction is closely related to the approach taken to defining the classification relationships in a deep model - constructive modeling or exploratory (a.k.a. explanatory) modeling [72]. When a model is being developed for a constructive purpose (e.g., to create a software system or a database), all the types are defined first, and then all the instances are created by instantiating them. The resulting instances are therefore bound to be isonyms of their direct types because they are created from the templates they define. However, when a model is being developed for an exploratory purpose, the instances exist first and the types are derived from them. In such cases, the modeler is usually only interested in capturing the important attributes of the existing instances in the type model and may leave some actual features unmodeled. Direct instances are by definition hyponyms when they include additional features as compared to their direct type.

Universal Isonymic Instantiation: In constructive modeling scenarios where all instantiation should be isonymic, it is useful to apply the "Universal Isonymic Instantiation" style to ensure that all models are well-formed and that no hyponymic instantiation occurs. This style basically states that all clabjects that have an ontological direct type must be an isonym of that type. In other words, all direct instances of a clabject must be isonyms of that clabject. This style is defined and enforced by Constraints 9.3 and 9.4

```
context Clabject(0,_)
inv: if self.doclGetDirectInstances() -> size > 0
    then let featureSize:Integer = self.#getFeature()# ->
    select(f|f.#getDurability()# > 0) -> size() in
    self.doclGetDirectInstances() -> forAll(i|i.#getFeature()# -> size() =
    featureSize)
    else true
    endif
```

CONSTRAINT 9.3: Universal isonymic instantiation style: instances have to have the same number of potent features as their types

```
context Clabject(0,_)
inv: if self.doclGetDirectInstances() -> size > 0
    then let featureNames:Set = self.#getFeature()# ->
    select(f|f.#getDurability()# > 0) -> collect(#name#) in
    self.doclGetDirectInstances() -> forAll(i|i.#getFeature()# ->
    collect(#name#) -> includesAll(featureNames))
    else true
    endif
```

CONSTRAINT 9.4: Universal isonymic instantiation style: instances have to have the same potent features as their types

It is of course possible to define the opposite "Universal Hyponymic Instantiation" style in which only hyponymic classification relationships are allowed in a model, but it is not clear whether there are any practical uses for this style.

9.3 Inheritance Styles

The styles presented in the previous section focused on statements about the relationship between clabjects and their direct types - that is, on the relationship between a clabject and its direct type. They clarify which clabjects in a model are mandated to have such relationships and what level of characterization they should provide. However, the aforementioned styles do not make any statements about, or define styles related to indirect classification.

Indirect classification is intimately tied to the notion of subtyping (a.k.a. specialization) or supertyping (a.k.a. generalization), where the former is the inverse of the latter. In fact, the very distinction between direct and indirect types relies on the existence of such relationships between clabjects. Since the relationship that captures the existence of supertype/subtype relationships between clabjects is called the inheritance hierarchies, in this section we refer to hierarchies of such relationships as inheritance relationships and call the styles that govern their form "inheritance styles".

9.3.1 Inheritance Declaration

The classification styles defined in the previous section describe whether or not direct typing relationships may exist between clabjects at different levels, and if they do exist, what form they should have (i.e., isonymic or hyponymic), but they do not prescribe whether particular pairs of clabjects should have a direct typing relationship or not. There are two basic strategies for allowing modelers to make this decision – so-called structural approaches and nominal approaches.

Structural Typing: In languages like LML that only allow clabjects to have a maximum of one direct type, structural typing approaches are based on four basic premises –

ST1 If a clabject, I, satisfies the structural requirements needed to be an instance of a clabject, T, (it is either a hyponym or isonym of T) I MUST be regarded as an instance of T (or, alternatively, T must be regarded as a type of I).

ST2 If the set of clabjects that must be regarded as instances of a clabject X (according to criteria **ST1**) is a proper subset of the set of clabjects that must be regarded as instances of clabject Y, then Y MUST be regarded as a being a supertype of X, possibly with intermediate types. In other words, X must be below Y in a chain of inheritance relationships,

ST3 If a clabject I must be an instance of T according to **ST1**, and there is no subtype of T that I must be regarded as being an instance of according to **ST1**, then T MUST be regarded as the direct type of I.

ST4 A clabject cannot have more than one direct type.

Note that to follow these rules, modelers often need to be creative. For example, if rules ST1, ST2, and ST3 leave two or more existing classes as potential direct types of a clabject, the modelers must introduce a new clabject that inherits from these clabjects (multiple inheritance) so that it can serve as the unique direct type.

Each of these rules can be defined using DOCL:

```
context Clabject(0,_)
inv: Clabject -> reject(self) ->
    exists(c|(self.doclIsHyponymOf(c) or self.doclIsIsonymOf(c))
    implies self.doclIsInstanceOf(c))
```

CONSTRAINT 9.5: Structural Typing: constraint for the ST1 rule

```
context Clabject(0,_)
inv: Clabject -> reject(self) ->
    select(c|c.doclGetDirectInstances() -> size() > 0 and
    c.doclGetInstances() -> includesAll(self.doclGetInstances())) ->
    forAll(c| if self.#getSubtypes()# -> size() = 0
        then true
        else self.#getSubtypes()# -> includes(c)
    endif)
```

CONSTRAINT 9.6: Structural typing: constraint for the ST2 rule

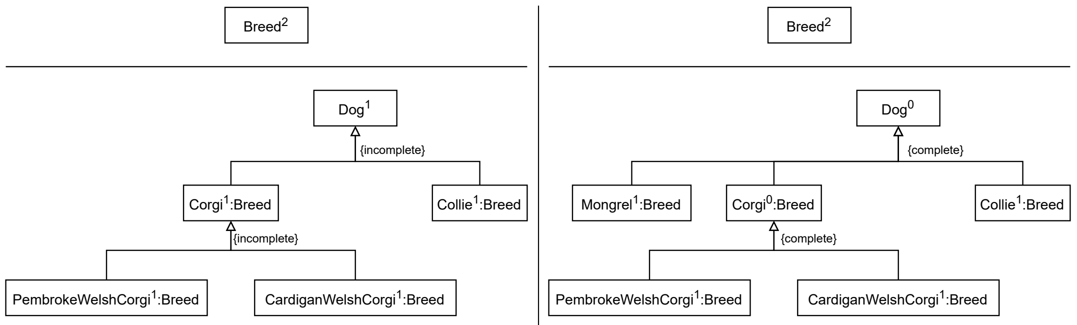
```
context Clabject(0,_)
inv: Clabject -> reject(self) ->
    select(c|(self.doclIsHyponymOf(c) or self.doclIsIsonymOf(c)) and
    c.#getSubtypes()# -> size() = 0) ->
    forAll(c|self.#getDirectType()# = c)
```

CONSTRAINT 9.7: Structural typing: constraint for the ST3 rule

```
context Clabject(0,_)
inv: self.#getDirectType()# -> size() <= 1
```

CONSTRAINT 9.8: Structural typing: constraint for the ST4 rule

Nominal Typing: Norminal typing differs from structural typing in that it gives the modeler the final say on whether a clabject that is an isonym or hyponym of another clabject should actually be regarded as an instance of that clabject, or whether two clabjects whose extensions satisfy **ST2** should be regarded as being in an inheritance relationship. In other words, in nominal typing, **ST1**, **ST2**, and **ST3** apply but with the capitalized word "MUST" in the previous definitions replaced by "MAY". Under the nominal modeling style, therefore, modelers are not obliged to accept direct or indirect type relationships that must be accepted under the structural style. In nominal typing "candidate" direct and indirect typing relationships that are possible according to structural typing have to be explicitly selected by the modeler and declared to be extant by adding corresponding elements to the model. In both cases, the modelers are responsible for introducing suitable clabjects



(A) Potent non-abstract superclass

(B) Abstract superclass

FIGURE 9.2: Forms of generalization sets (inspired by [77])

and relationships to ensure that the requirements of other styles are met, if they have been selected (e.g., the Universal Classification Style).

9.3.2 Specialisation Set Form

Specialization sets[1] play an important role in all forms of object-oriented modeling since they show when subtyping relationships are related to one another, and how the extensions of clabjects are split up into the extensions of other clabjects. They have two concrete properties, disjointness, and completeness, which can interact in various with the supertype (whether or not the supertype is abstract). This interaction naturally leads to two styles of usage of both properties.

The interaction is illustrated in Figure 9.2 which shows two alternative ways of representing the same domain scenario – namely, the situation where a pet shop sells a certain set of pure Dog breeds as well as dogs that do not belong to any breed. Figure 9.2a shows how this scenario can be modeled using an incomplete generalization set in which the pure breeds are concrete subtypes of a concrete supertype, *Dog*. Pure breed dogs (i.e., instances) are thus direct instances of their respective breed clabjects, while non-pure breed dogs (often called Mongrels in English) have *Dog* as their direct type. Figure 9.2b shows the same domain scenario but uses a complete specialization set with an abstract supertype, *Dog*, and a new concrete subtype *Mongrel*. The essential difference is that non-pure breed dogs now have Mongrel as their direct type rather than Dog.

Hürsch [62] pointed out that these alternatives exist for all specialization sets, in the context of object-oriented programming, and that it is possible to systematically translate a program using concrete supertypes into another

[1]LML uses the term specialization set for the UML notion of a generalization set

program using only abstract supertypes without changing the behavior of the resulting program. Based on this observation, they proposed the so-called "Abstract Superclass Rule" which advocates that the style depicted in Figure 9.2b be used. While the rule is not intended to be an indicator of a good or bad program per se, it has certain advantages, such as reusability and simplicity, which are considered beneficial for object-oriented frameworks and libraries. when one of the two aforementioned approaches is or should be, applied exclusively over one or more levels, they give rise to corresponding styles.

Exclusively Abstract Supertypes: The "Exclusively Abstract Supertypes" style essentially exclusively applies the approach shown in Figure 9.2b throughout one or more levels and prohibits the use of the approach shown in Figure 9.2b. The implication of having this style in place is that every specialization set has to be complete. In other words, concrete (non-abstract) subclasses must be leaves of an inheritance hierarchy and partition the domain of the generalization completely so that every object that needs an ontological type can be characterized by one of the concrete subclasses.

```
context Inheritance(0,_)
inv: self.#getSupertypes()# -> forAll(s|s.#getPotency()# = 0 and
     s.doclGetDirectInstances() -> size() = 0) and
     self.#isComplete()# = true
```

CONSTRAINT 9.9: Exclusively abstract supertypes style

Constraint 9.9 shows that this style can be enforced relatively simply using DOCL. The context here is the *Inheritance* class and, for all instances, it must be true that the *supertype* has a potency of 0 (i.e., is an abstract clabject) and the inheritance must have the property *complete*.

Exclusively Concrete Supertypes: This style is the opposite of the previous style since it demands that every superclass in the specialization set must be concrete (i.e., not abstract). This style offers more flexibility for classifying objects.

```
context Inheritance(0,_)
inv: self.#getSupertypes()# -> forAll(super|super.#getPotency()# > 0) and
     self.#isComplete()# = false
```

CONSTRAINT 9.10: Exclusively concrete supertypes style

Constraint 9.10 shows a very similar constraint to Constraint 9.9 that states that every superclass in an inheritance structure has to have a potency value greater than 0 and the generalization set has to be incomplete.

Universal Root Inheritance: This style requires all levels to have one clabject that is the root of all other clabjects in that level except possibly the most concrete level that contains the instances that model individuals in the domain. This obviously requires inheritance to be possible within a level, which is not the case if the clabjects at the bottom exclusively represent individuals in the real world. This style can be activated for all levels, a level range, or just one level. The context of the constraint is *Level* which means that the constraint has to be valid for every linguistic instance of the *Level* class.

```
context Level(0,_)
inv: if self.#isLeafLevel()#
    then true
    else self.#getClabjects()# -> select(c|c.#getSupertypes()# ->
    size() = 0) -> closure(c|c.#getSubtypes()#)
    -> includesAll(self.#getClabjects()#)
    endif
```

CONSTRAINT 9.11: Universal root inheritance style

9.4 Vitality Styles

A key element of the core style presented in Chapter 8 is the characterization potency rule, defined in Constraint 8.4, which states that the potency of a direct instance of a clabject must be lower than that clabject. This was introduced to deal with situations like the one shown in Figure 9.2, where it seems natural for abstract and concrete clabjects in an inheritance hierarchy to be direct instances of the same clabjects.

However, this choice entails trade-offs that may not be optimal for all domains and use cases. There are various alternative options for dealing with the situation in Figure 9.2. The downside of classification potency is that it leaves the modeler with the decision about what potency a direct instance of a clabject should have. Although the potency of a direct instance is constrained to be lower, the modeler has to decide which of the allowed potency values to choose, which can lead to errors.

Classic Potency Style: The classic potency style essentially reinstates the original approach to deep instantiation defined in [19] by stating that the potency of a direct instance of a clabject must be exactly one lower than the potency of that clabject. Constraint 9.12 encodes this rule in DOCL.

```
context Clabject(0,_)
inv: self.doclGetDirectInstances() ->
    forAll(c|c.#getPotency()# = self.#getPotency()# - 1)
```

CONSTRAINT 9.12: Classis potency style

Note that this rule is compatible with the core style because it strengthens the characterization potency rule. However, its application may in certain circumstances force modelers to make other choices for the clabjects involved in a generalization. For example, the domain scenario depicted in Figure 9.3, which was the example used to motivate the deep potency variant in [77], forces certain combinations of choices to be made if the classic potency style is applied.

The key clabject that demonstrates the issues involved is the *Corgi* clabject in the *Dog* generalization set. When the more relaxed, characterization potency style is applied, *Corgi* can be a direct instance of *Breed* (like *Mongrel* and *Collie*) whilst still being abstract. A potency value of 0 is acceptable in this case but classic potency rules would demand *Collie* to have a potency value of 1. If the subclasses *PembrokeWelshCorgi* and *CardiganWelshCorgi* were not present in the model, this would certainly be the most natural direct type for *Corgi*. However, when present, other choices are also justifiable. One option is to make it a direct type of another clabject which "characterizes" *Dog*, using the approach of MLT [29] (see Section 10.3.1), and another, simpler option is to not assign an ontological type to *Corgi*, as in Figure 9.3. This allows *Corgi* to have any potency, suitable for representing its classification role (e.g., abstract, concrete, etc.) whilst still adhering to the classic potency rule. In order to ensure that these two options are adhered to throughout a deep model or at least within a level, it is possible to define corresponding styles.

Untyped Abstract Types Style: As mentioned above, the easiest way to make sure that all generalization sets in a deep model adhere to the classic potency rule, whilst accommodating nested complete generalization sets such as that depicted in Figure 9.3, is to ensure that all abstract types are ontologically untyped. This makes it impossible for the situation captured by Figure 9.3 to arise.

The constraint that defines this "Untyped Abstract Types" style is Constraint 9.13. A clabject is abstract when the potency value is 0 and it participates in an inheritance relationship as a superclass. Note that this style could be combined with the Exclusively Abstract Supertypes style, but does not make sense in combination with the Exclusively Concrete Supertypes

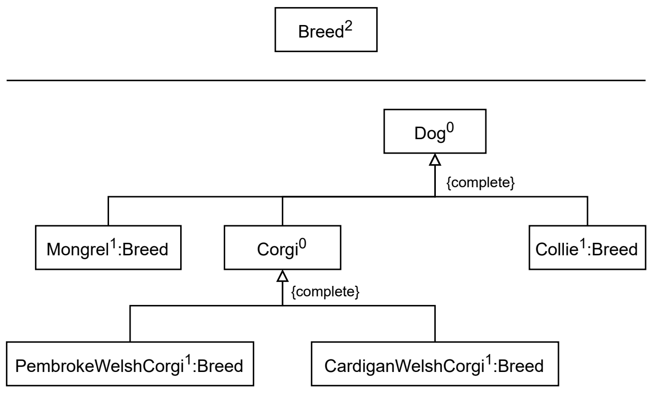


FIGURE 9.3: Untyped Abstract Types Style

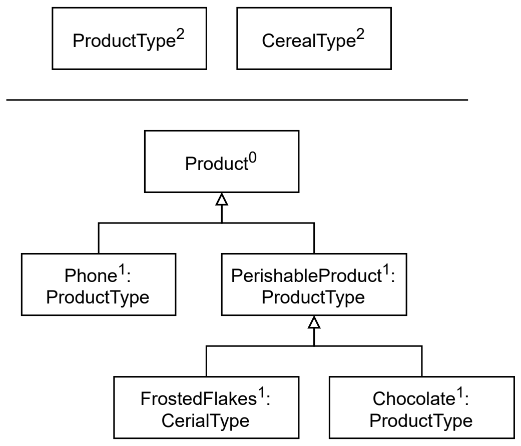

ProductType^2
CerealType^2
Product^0
Phone^1: ProductType
PerishableProduct^1: ProductType
FrostedFlakes^1: CerialType
Chocolate^1: ProductType

FIGURE 9.4: Harmonious Horizontal Superclasses example

style (the two styles could be applied together, but then the Untyped Abstract Types style would have no effect).

```
context Clabject(0,_)
inv: if self.#getPotency()# = 0 and self.#getSubtypes()# -> size() > 0
        then self.#getDirectType()# -> isEmpty()
        else true
        endif
```

CONSTRAINT 9.13: Untyped abstract types style

Harmonious Horizontal Supertypes: This style enforces the other approach for accommodating nested complete generalization sets in combination with the classic potency style mentioned above. It basically requires all direct instances of a clabject to be on the same subclass level if they are taking part in an inheritance relationship, and all the clabjects at the same subclass level must have the same direct ontological type. In other words, clabjects that participate in inheritance relationships involving generalization sets have to be at the same depth of the inheritance structure.

Figure 9.4 shows an example of this style in play. The nested generalization sets shown in the diagram are not allowed by the style because the clabject *ProductType* has instances at two different levels in the inheritance hierarchy, *Phone* and *PerishableProduct* at the second inheritance level and *Chocolate* at the third level.

```
context Clabject(0,_)
inv: let type:Clabject = self.#getDirectType()# in not (self.#getSupertypes()#
    -> exists(s|s.doclIsDeepTypeOf(type)))
```

CONSTRAINT 9.14: Harminious horizontal supertypes style: at a given level in a generalization set subtypes have to have the same direct type

Constraint 9.14 shows the DOCL constraint to enforce the style in all instances of *Inheritance*. In this constraint, we first set the type of the first subclass as the type to compare every other subclass with. It has to be true that for all of the subclasses of this particular generalization set at the same level, all subclasses must have the same type as the first subclass in the set.

```
context Clabject(0,_)
inv : let type:Clabject = self.#getDirectType()# in
    not (self.#getDirectSupertypes()# -> exists(s|s.doclIsDeepTypeOf(type)))
```

CONSTRAINT 9.15: Harminious horizontal superclass style: types must be mutually exclusive at every supertype level

Constraint 9.15 shows how the supertype of a clabject must be different from the type of the clabject in question. There must not exist a clabject that is a supertype of another clabject where both have the same ontological type. The combination of both constraints ensures that all the clabjects at a given subclass level in an inheritance hierarchy (a) have the same direct ontological types and (b) have a different ontological type to the clabjects in all the other levels in the same inheritance hierarchy.

9.5 Cross-Level, Well-Formedness Styles

The strict modeling style described in Section 8.3 was designed to rule out numerous anti-patterns that can occur in multi-level modeling [18]. However, as pointed out by [78], for some modeling scenarios, it can be too strict, leading to the need for additional redundant classes. This is the reason why most MLM languages and tools do not apply the strict modeling style in its extreme form. However, when the strict modeling style is not active, there is no guarantee that modelers will not make errors. This is particularly the case

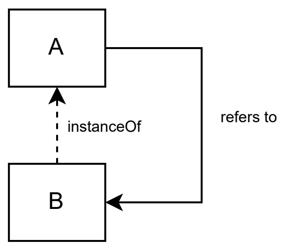


FIGURE 9.5: A Metacycle [18]

when there are no restrictions about how clabjects at one level can be related to clabjects at another level.

Partly to motivate the strictness style, Atkinson and Kühne [18] introduced three fundamental anti-patterns that should never occur in multi-level models - the metacycle antipattern, the metabomb antipattern, and the type-supertype anti-pattern. The strict modeling pattern rules these out with a few simple, but "strict", constraints. However, to gain the same benefits without the downside of strict modeling constraints, it is possible to define separate, dedicated styles focused on each pattern individually. As with the Orderly Hierarchy styles in section 8.2.1, each of the separate styles essentially rules out the occurrence of one of the anti-patterns. Modelers can then activate the styles in the combination that best matches their particular modeling scenario.

Absence of Metacycle Style: One of the three cross-level anti-patterns that can occur if the strict modeling style is not active is the metacycle anti-pattern. As shown in Figure 9.5, this occurs when a clabject *A* is the type of another clabject *B* (i.e., *B* is a direct instance of *A*), whilst at the same time having a connection to **B**. In this example, the "refers to" relationship between *A* and *B*, where *B* is a direct instance of *A* is an occurrence of the Metacycle anti-pattern.

```
context Clabject(0,_)
inv: self.#getClassificationTreeAsInstance()# ->
    closure(c|c.#getAllNavigations()#.#destination#) -> excludes(self)
```

CONSTRAINT 9.16: Absence of Metacycle style

Occurrences of this anti-pattern can be ruled out by applying the Absence of Metacycles style which is defined in DOCL by the Constraint 9.16. This is a reflective constraint that queries for all offspring of a clabject (i.e., using *closureWithoutSelf*), gets the direct type of the clabject, and unions the resulting collection with all navigation destinations from that direct type. This operation continues until no new elements can be reached or if a circle is detected.

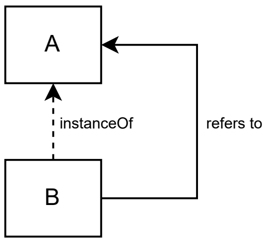


FIGURE 9.6: A Metabomb [18]

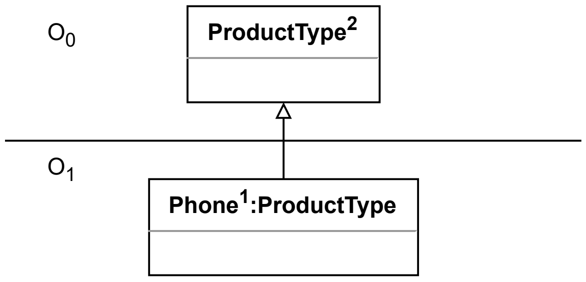


FIGURE 9.7: Type-Supertype Anti-Pattern

Absence of Metabomb Style: The second cross-level, anti-pattern described in [18] is the "Metabomb" anti-pattern depicted in Figure 9.6. The instance of relationship is the same as in the previous example, but in this case the "refers to" relationship is from the instance *B* to its direct type *A*. This anti-pattern is characterized by the authors as being less problematic than the Metacycle anti-pattern but it nevertheless should be avoided. Occurrences of this anti-pattern can be ruled out by applying the Absence of Metabombs style which is defined in DOCL by the Constraint 9.17.

```
context Clabject(0,_)
inv: self.#getClassificationTreeAsInstance()# ->
     excludesAll(self.#getAllNavigations()#.#destination#)
```

CONSTRAINT 9.17: Absence of Metabombs style

In this case, the same *nonReflexiveClosure* operation is used to traverse the classification hierarchy in the upward direction, and none of the elements of this collection must be in the collection that can be reached via any connection from the clabject itself.

Type-Supertype Anti-Pattern: The third anti-pattern identified by Atkinson and Kühne [18] is the Type-Supertype pattern depicted in Figure 9.7. This pattern can emerge in MLM approaches that allow inheritance relationships as well as connections to cross ontological levels. In this example, the *Phone* clabject is a subclass of *ProductType* and also a direct instance of it.

Occurrences of this anti-pattern can be ruled out by applying the Absence of Type-Supertype Anti-pattern style which is defined in DOCL by the Constraint 9.18. Due to the general nature of the anti-pattern, the context of the constraint is the *Clabject* class of the linguistic meta-model. The set of superclasses must not overlap with the set of types in the ontological classification hierarchy. In other words, there must not be a superclass of any clabject that is simultaneously a type of that clabject.

```
context Clabject(0,_)
inv: self.#getDirectType()# -> closure(c|c.#getTypes()#) ->
    excludesAll(self.#getSupertypes()#)
```

CONSTRAINT 9.18: Absence of the Type-Supertype Anti-Pattern

Chapter 10

Multi-Pattern Modeling

A key feature of the deep modeling styles described in the previous section is that they cover all the model elements in at least one level, and usually more levels, and they do so in an ontologically blind way. In other words, they are defined exclusively in terms of the linguistic classifiers of the model elements in the ontological dimensions.

Modeling patterns are different in two ways. First, they generally have a smaller scope and involve a relatively small group of model elements. Second, they relate the involved model elements based on both their ontological and linguistic facets, rather than purely using their linguistic facet. The main consequence of these differences is that declaring the presence of a pattern involves identifying the domain model elements that are meant to be involved.

The classic example is the powertype pattern. This pattern relates model elements in adjacent levels of a multi-level model and thus occurs between a pair of levels. However, it usually only occurs selectively based on the domain concepts that are being modeled. To identify, and subsequently check, the well-formedness of patterns it is necessary to identify the concepts in the domain that are supposed to be in the pattern.

Note that patterns, anti-patterns, and styles are related. First, styles may automatically prohibit certain patterns and anti-patterns from existing in a model. For example, the strict modeling style rules out the circular classification and inheritance anti-patterns. Second, styles can be defined by the universal application of patterns or the universal prohibition of anti-patterns. For example, the exclusively abstract supertypes style can be defined by universally applying the complete discrimination pattern (see Section 10.2.1), while the absence of a circular classification style can be defined by universally prohibiting the circular classification anti-pattern.

10.1 Classification Patterns

As explained in the previous chapter, the difference between styles and patterns is that the former defines rules that apply across one or more levels and are defined solely in terms of the linguistic types (i.e., in the PLM), while the latter defines rules that apply to specific groups of clabjects and have to be explicitly associated with the domain clabjects involved. The size of the group of clabjects can range from very small, with a minimum size of two, to much larger groups crossing multiple levels. This section starts with a set of small, "fundamental" patterns that essentially define the meaning of terms often used in conceptual modeling. Although they may seem somewhat trivial their main value is ensuring that the intended properties of the domain element(s) involved are preserved as a model changes.

10.1.1 Impotent Clabject Patterns

The first two patterns formalize the two different interpretations of impotent (i.e., potency 0) clabjects that can occur in LML models. As well as clarifying the semantics of these interpretations, the defined constraints ensure that the associated rules are preserved as a model evolves.

Abstract Clabject Pattern: The first role that impotent clabjects can play in LML models is that of abstract classes. Although abstract classes can have instances, by virtue of their subclasses, they can only have indirect instances. Since the potency value of a clabject is defined in terms of its direct instances, abstract classes have a potency value of 0. In order to qualify as an abstract clabject, a clabject must (a) have no direct instances in any world and (b) have at least one subclass. These rules are formalized in Constraint 10.1.

```
context AnAbstractClabject(0,0)
inv: self.#getSubtypes()# -> size() > 0 and self.#getPotency()# = 0
    and self.doclGetDirectInstances() -> size() = 0
```

CONSTRAINT 10.1: Abstract clabject pattern

Individual Clabject Pattern: The second role that impotent clabjects play in LML models is that of individuals. An individual is a clabject that does not have and can never have (in any possible world), any instances, either direct or indirect. It therefore follows that individuals cannot participate in inheritance relationships. These rules are formalized in Constraint 10.2.

```
context AnIndividualClabject(0,0)
inv: self.#getSupertypes()# -> size() = 0
    and self.#getSubtypes()# -> size() = 0
    and self.#getPotency()# = 0
    and self.doclGetDirectInstances() -> size() = 0
```

CONSTRAINT 10.2: Individual Clabject Pattern

10.1.2 Instance Location and Occurrence Patterns

The previous pattern defines rules that completely rule out the presence of instances. In contrast, the two patterns presented in this subsection deal with the presence and/or location of instances when they do occur.

Singleton Clabject Pattern: This a well-known pattern from the Gang of Four [52] which is not only commonly used in software engineering, but also captures important domain concepts, such as the leader of an organization (e.g., there must be, and can only be, one president of an organization). This pattern applies when there is always, in all possible worlds, one and only one instance of a clabject. Constraint 10.3 formulates this fact.

```
context ASingletonClabject(0,0)
inv: self.doclGetDirectInstances() -> size() = 1
```

CONSTRAINT 10.3: Singleton Clabject Pattern

Classic Potency Pattern: Section 9.4 of the previous chapter presented the Classic Potency Style, in which the original rules of potency are applied. In contrast to characterization potency, which only requires the direct instances of a clabject to have a lower potency than their own potency, the classic potency style requires the direct instances of a clabject to have a potency that is exactly one lower than their own. Since potency cannot take a value less than zero, this means that the offspring of a clabject occupy lower ontological levels in an orderly manner – the clabjects whose potency is one lower must occupy the level below, the clabjects whose potency is two lower must occupy the second level below, and so on up to the impotent clabjects which occupy the nth level below, if the potency of the clabject concerned is n.

When a model is being created using the characterization potency style it may still be useful to define localized offspring hierarchies, originating from a single clabject, that adheres to the rules of classic potency so that the offspring with the same potencies occupy the same levels. Constraint 10.4 defines this pattern. Note that it is redundant (i.e., has no effect) if the classic potency style is applied. Since classic potency is a special case of characterizing potency, it is acceptable for certain offspring hierarchies to conform to the classic potency rules while the characterization style is in effect for the whole level or model (e.g., as part of the default style).

```
context AClabject(0,0)
inv: self.doclGetDirectOffspring() ->
    iterate(offspring; currentPotency:Integer = self.#getPotency()#|
        currentPotency - offspring.#getPotency()#) = 1
```

CONSTRAINT 10.4: Classic potency pattern

10.2 Inheritance Patterns

The previous section presented patterns focusing on the presence, absence, or location of classification relationships. This section presents patterns whose focus is on inheritance relationships, and more specifically, inheritance relationships which form part of what in the UML are called generalization sets.

10.2.1 Discrimination Patterns

In the UML, a generalization set combining one superclass (the generalization) with multiple subclasses (the specializations) is said to “discriminate” between the instances of the superclass according to some concern, which is known as the “discriminator”. We, therefore, refer to patterns that apply to generalization sets as discrimination patterns. Given the terms “generalization” and “specialization” refer to the same relationship, in this thesis, we prefer to refer to generalization sets as “specialization sets” since it is the specialization of generalization sets that can form arbitrarily sized sets. There is always exactly one generalization class involved in a set, and thus the generalization (i.e., superclass) is always a single value set.

Note that in the LML, specialization sets are represented by a single instance of the linguistic class *Inheritance*. According to the PLM, an instance of this class can have links to multiple clabjects that play the role of superclasses and multiple clabjects that play the role of subclasses. When representing a specialization set, however, there must be exactly one clabject playing the role of the superclass, and one or more classes playing the role of a subclass.

Basic Discrimination Pattern: A discrimination pattern divides the extension of a clabject into multiple subsets. This is modeled by means of a specialization set (represented as an instance of the class *Inheritance* from the PLM) with one superclabject and multiple subclabjects. As a minimalist case, it is possible for a specialization set to divide the extension of a clabject into two groups even if it only contains one subclass as long as the superclabject is abstract. Although one of the subsets is anonymous in such a scenario (i.e., the group of instances that are direct instances of the superclabject), the membership of that subset is nevertheless well-defined. The definition of this pattern in Constraint 10.5 therefore takes both these cases into account.

```
context AnInheritance(0,0)
inv: (self.#getSupertypes()# -> size() = 1 and
    self.#getSupertypes()#.#getPotency()# > 0) or
    self.#getSubtypes()# -> size() >= 2
```

CONSTRAINT 10.5: Basic Discrimination pattern

Complete Discrimination Pattern: Like UML generalization sets, LML specialization sets have two properties that define whether the subsets of the extension of the superclabject are "complete" and/or "disjoint". The basic discrimination pattern defined above makes no statement about these properties and thus allows any combination of the two. In contrast, complete discrimination strengthens the basic discrimination pattern defined above by requiring the first of these properties to be true, without making a statement about the second. Constraint 10.6 formalizes this.

```
context AnInheritance(0,0)
inv: let type: Clabject = self.#getSupertypes()# -> first() in
    self.#getSupertypes()# -> size() = 1 and self.#getSupertypes()# ->
    forAll(s|s.#getPotency()# = 0) and
    self.#getSubtypes()# -> size() >= 2 and
    self.#getSubtypes()#.doclGetInstances() ->
    forAll(i|i.doclIsInstanceOf(type))
```

CONSTRAINT 10.6: Complete discrimination pattern

Disjoint Discrimination Pattern: The disjoint discrimination strengthens the basic discrimination pattern by requiring the second of the above properties to be true, without making a statement about the first. In other words, it requires the subsets of the extension of the superclabject (i.e., the discriminated clabject) to be disjoint but does not require them to be complete. Constraint 10.7 shows this.

```
context AnInheritance(0,0)
inv: self.#getSupertypes()# -> size() = 1 and
    self.#getSupertypes()# -> first().#getPotency()# = 0 and
    self.#getSubtypes()# -> size() >= 2 and
    self.#getSubtypes()# -> forAll(s|s.#getInstances()# ->
    excludesAll(self.#getSubtypes()# -> excluding(s).#getInstances()#))
```

CONSTRAINT 10.7: Disjoint discrimination pattern

Partitioned Discrimination Pattern: The partitioned discrimination pattern goes one step further by requiring a specialization set to be both complete and disjoint. This is defined in Constraint 10.8.

```
context AnInheritance(0,0)
inv: self.#getSupertypes()# -> size() = 1 and
    self.#getSupertypes()# -> first().#getPotency()# = 0 and
    self.#getSubtypes()# -> size() >= 2 and self.#getSubtypes()# ->
    forAll(s|s.#getInstances()# -> excludesAll(self.#getSubtypes()# ->
    excluding(s).#getInstances()#))
```

CONSTRAINT 10.8: Partitioned discrimination pattern

10.3 Categorization Patterns

The discrimination patterns defined in the previous section are all essentially classic “two-level” patterns since they only consider the relations between types and their immediate instances at the level below. However, an important area of research in MLM are patterns that relate discrimination patterns to the types of the types involved.

The most well-known and widely researched example is the so-called powertype pattern which has been discussed in the literature for many years and arguably has played a role in kicking off the field of MLM [20, 29, 56]. This section discusses several powertype-related patterns based on the terminology and definitions supported by the MLT approach [29].

10.3.1 Weak Categorization Patterns

There are two groups of categorization patterns, weak categorization patterns and strong categorization patterns. For each weak categorization pattern, there is a corresponding strong categorization pattern. The stronger versions all have the same extra rule that prohibits instances of clabjects other than the powertype (or characterizing) clabject from participating as subclasses in the respective specialization sets.

The first MLM approach to propose a systematic set of weak categorization patterns was MLT, but they use the term "characterization patterns". For consistency with the MLT terminology we, therefore, also refer to the various weak categorization patterns as characterization patterns.

Basic Characterization Pattern: Just as four different forms of discrimination patterns were defined in Section 10.2.1 according to the combinations of the disjoint and complete properties they require. In the same way, four different forms of characterization patterns can be defined based on the kind of discrimination patterns they enforce.

The archetypal characterization pattern is the powertype pattern proposed by Odell [95] which states that Power(A) is the powertype of A if the instances of Power(A) are subtypes of A. This is referred to as the characterization pattern in MLT. We refer to it as the basic discrimination pattern.

In terms of the discrimination patterns, defined in the previous section, this basic characterization pattern essentially requires the instances of Power(A) to be the subclabjects in a basic discrimination pattern where A is the discriminated (i.e., super) clabject. When they are related by such a pattern, A is said to be the base type, or the characterized type, while Power(A) is said to be the powertype or the characterizing type. In DOCL, the basic characterization pattern is defined by Constraint 10.9.

```
context Power(A)(0,0)
inv: let inheritance:Inheritance = self.doclGetDirectInstances() ->
    first().#getInheritancesAsSubtype()# -> first() in
    let baseType:Clabject = inheritance.#getSupertypes()# -> first() in
    self.doclGetDirectInstances() ->
    forAll(i|i.#getInheritancesAsSubtype()# -> first() = inheritance)
```

CONSTRAINT 10.9: Basic Characterization pattern

It is common, although not mandatory, to give the discriminant in the specialization set the same name as the powertype.

Complete Characterization Pattern: The complete characterization pattern strengthens the basic characterization pattern by requiring the instances of Power(A) to be subclabjects in an application of the complete discrimination pattern to A. In DOCL this is represented by the Constraint 10.10.

```
context Power(A)(0,0)
inv: let inheritance:Inheritance = self.doclGetDirectInstances() ->
     first().#getInheritancesAsSubtype()# -> first() in
     let baseType:Clabject = inheritance.#getSupertypes()# -> first() in
     self.doclGetDirectInstances() -> forAll(inst|
     inst.#getInheritancesAsSubtype()# = inheritance) and
     baseType.#getPotency()# = 0 and
     baseType.#getSubtypes()#.doclGetDirectInstances() ->
     forAll(inst|inst.doclIsIndirectInstanceOf(baseType))
```

CONSTRAINT 10.10: Complete characterization pattern

Disjoint Characterization Pattern: The disjoint characterization pattern strengthens the basic characterization pattern by requiring the instances of Power(A) to be subclabjects in an application of the disjoint discrimination pattern to A. In DOCL this is represented by the Constraint 10.11.

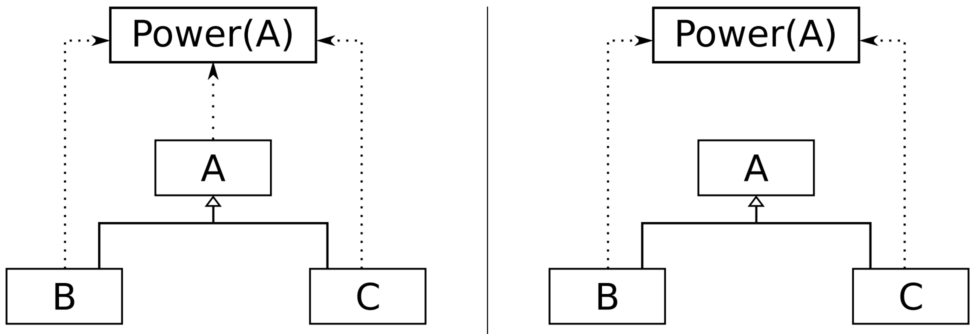
```
context Power(A)(0,0)
inv: let inheritance:Inheritance = self.doclGetDirectInstances() ->
     first().#getInheritancesAsSubtype()# -> first() in
     let baseType:Clabject = inheritance.#getSupertypes()# -> first() in
     self.doclGetDirectInstances() ->
     forAll(inst|inst.#getInheritancesAsSubtype()# -> first() = inheritance) and
     baseType.doclGetInstances() ->
     forAll(baseInst|baseType.#getSubtypes()# ->
     excluding(baseInst.#getDirectType()#) ->
     (not exists(sub|baseInst.doclIsInstanceOf(sub))))
```

CONSTRAINT 10.11: Disjoint characterization pattern

Partitioned Characterization Pattern: The partitioned characterization pattern strengthens the basic characterization pattern by requiring the instances of Power(A) to be subclabjects in an application of the partitioned discrimination pattern to A. In DOCL this is represented by the Constraint 10.12.

```
context Power(A)(0,0)
inv: let inheritance:Inheritance = self.doclGetDirectInstances() ->
     first().#getInheritancesAsSubtype()# -> first() in
     let baseType:Clabject = inheritance.#getSupertypes()# -> first() in
     self.doclGetDirectInstances() ->
     forAll(inst|inst.#getInheritancesAsSubtype()# -> first() = inheritance) and
     baseType.doclGetInstances() ->
     forAll(baseInst|baseType.#getSubtypes()# ->
     excluding(baseInst.#getDirectType()#) ->
     (not exists(sub|baseInst.doclIsInstanceOf(sub)))) and
     baseType.#getPotency()# = 0
```

CONSTRAINT 10.12: Partitioned characterization pattern



(A) Cardelli powertype pattern (B) Odell powertype pattern

FIGURE 10.1: Powertype Patterns

10.3.2 Strong Categorization Patterns

The difference between strong and weak categorization patterns is that the former strengthens the latter by ruling out instances of other clabjects from being subclasses in the involved discrimination of the base clabject. It is possible to define a strong variant of each of the weak characterization pattern variants defined in the previous section. However, for illustration purposes, in this section only one variant is shown, the strong Odell Powertype Pattern. This is a strong version of the standard Odell Powertype Pattern (a.k.a, the basic characterisation pattern). The other four characterization patterns can be strengthened in the same way.

Strong Odell Powertype Pattern: The reason why the characterization patterns presented in the previous section are referred to as weak is that although they require all the instances of the powertype, Power(A), to be subclabjects in an application of a discrimination pattern to the base type A, they do not prohibit instances of other clabjects, or non-ontologically typed instances, from participating in that pattern as subclabjects. The strong versions of the characterization patterns prohibit such participation by strengthening the constraints. In the case of the strong Odell powertype pattern, the Constraint 10.13 formalizes this.

```
context Power(A)(0,0)
inv: let inheritance:Inheritance = self.doclGetDirectInstances() ->
    first().#getInheritancesAsSubtype()# -> first() in
    let baseType:Clabject = inheritance.#getSupertypes()# -> first() in
    self.doclGetDirectInstances() ->
    forAll(inst|inst.#getInheritancesAsSubtype()# ->
    first() = inheritance) and baseType.#getSubtypes()# ->
    forAll(sub|sub.doclIsInstanceOf(self))
```

CONSTRAINT 10.13: Strong Odell powertype pattern

Cardelli Powertype Pattern: The first powertype pattern to actually include such a "strong" exclusion rule was the Cardelli Powertype, defined a few years before the Odell powertype pattern [27]. However, it also differs from the weak categorization patterns presented in Section 10.3.1 in another important way, as illustrated in Figure 10.1a. According to Cardelli, for a clabject Power(A) to be the powertype of another clabject A, the extension of Power(A) not only includes the subclabjects of A, but also A itself, and only instances of Power(A) can be subclabjects of A. This is why the Cardelli Powertype pattern is not regarded as a characterization pattern by MLT. The DOCL constraint defining this pattern is shown in the Constraint 10.14.

```
context Power(A)(0,0)
inv: let inheritance:Inheritance = self.doclGetDirectInstances() ->
    reject(inst|inst.#getSupertypes()# -> size() = 0) ->
    first().#getInheritancesAsSubtype()# -> first() in
    let baseType:Clabject = inheritance.#getSupertypes()# -> first() in
    self.doclGetDirectInstances() ->
    reject(inst|inst.#getSupertypes()# -> size() = 0) ->
    forAll(inst|inst.#getInheritancesAsSubtype()# -> first() = inheritance) and
    baseType.doclIsInstanceOf(self) and
    baseType.#getSubtypes()# -> forAll(sub|sub.doclIsInstanceOf(self))
```

CONSTRAINT 10.14: Cardelli powertype pattern

Part V

Evaluation

The fifth part of the thesis contains four chapters that evaluate the utility of the styles and patterns presented in the previous part by showing how several of them were used to solve the challenges defined by the Multi workshop community over the last few years. These "are designed as benchmark modeling scenarios that aim to support objective comparisons between multi-level modeling approaches, allow technologies to demonstrate their abilities, stress-test technologies in order to expose potential weaknesses, and deepen the mutual understanding of approaches" [1].

1. **The Bicycle Challenge**: deals with the representation of different configurations of bicycles based on their components (such as frame, handlebar, and wheels) and attributes (such as serial number, purchase price, and sale price [2]).
2. **The Process Challenge**: deals with the representation of universal properties of process types along with task types, artifact types, actor types, and their various relations and attributes. It also involves an application of this model in the scope of a particular software engineering process [8].
3. **The Collaborative Challenge**: deals with the representation of companies, factories, produced devices, and owned artifacts, such as intellectual property [92].
4. **The Warehouse Challenge**: deals with the representation of product copies, product specifications, and product specification types with a particular emphasis on how to guarantee certain properties at the product level without fully determining them at higher levels [80].

Each chapter illuminates how DOCL was used to make the models more precise and concise, including the use of modeling styles and patterns where appropriate.

Chapter 11

The Bicycle Challenge

The MULTI 2018 Bicycle Challenge was the first of the challenges defined by the MULTI community as a vehicle for evaluating and comparing multi-level modeling approaches [2]. It deals with the representation of different configurations of bicycles based on their components (such as frame, handlebar, and wheels) and attributes (such as serial number, purchase price, and sale price). The solution has four levels and uses the Melanee default style defined in Section 8.3, but overrides the characterization potency style with the classic potency style (Constraint 9.12). This chapter is based on the solution to the Bicycle Challenge that was published in [84].

11.1 Requirements

The requirements for this challenge are split into two categories, one mandatory category and one optional category part. The mandatory requirements are here denoted with an "M" and the optional requirements with an "O".

M1) A configuration is a physical artifact that is composed of components. A component may be composed of other components or of basic parts.

M2) A component has properties such as weight, size, colour and unique serial number.

M3) A bicycle is built of components like a frame, a fork, two wheels and so forth, each of which is a component. The front wheel and rear wheel must have the same size.

M4) There are different categories of bicycles, such as mountain bike, city bike or racing bike, for different purposes such as mountain biking, city biking or racing. A racing bike is not suitable for tough terrain is suitable for races. It can be used in cities, too.

M5) Each category is further associated with some constraints, for example:

1. Every category of bicycle except for racing bikes may be equipped with an electric motor. Electric bikes need enforced brakes and a battery.
2. A mountain bike or a city bike may have a suspension.
3. A mountain bike may have a rear suspension. That is not the case for city bikes.
4. A racing bike has a racing fork and racing frame.
5. A racing fork does not have a suspension. or mud mount.
6. A racing frame is specified by top tube length, down tube length, and seat tube length.
7. A racing frame is made of steel, aluminum, or carbon.
8. A racing bike can be certified by the Union Cycliste Internationale (UCI).
9. A professional racing bike is a racing bike and certified by the UCI. A professional racing bike has a professional race frame which is made of aluminum or carbon and has a minimum weight of 5200 gr.
10. A carbon frame allows for carbon wheels or aluminum wheels only.

M6) Each category of bicycle is associated with a person acting as a category manager. Peter Parker is the category manager for the racing bike category.

M7) Challenger A2-XL is a professional racing bike model for tall cyclists. A Challenger A2-XL bike is equipped with a Rocket-A1-XL which is a professional racing frame. The Rocket-A1-XL has a weight of 920.0 g.

M8) Bike#134123, a physical instance of Challenger A2-XL, has Frame#134123, a physical instance of Rocket-A1-XL with serial number s134123 as a component.

M9) Bicycles are sold to customers. A customer is a natural person or an organization. The act of selling a bicycle requires the creation of an invoice. An invoice is a read-only business document.

M10) Each bicycle model has a regular sales price. The regular sales price of Challenger A2-XL is EUR 4999.00. The actual sales price of physical instances of the bicycle model, i.e., the price given in an invoice, may be lower.

M11) Bike#134123 was sold on September 19th, 2017 for EUR 4299.00 to customer Susan Storm.

O1) The average actual sales price for a bike model. For example: In 2017, physical instances of Challenger A2-XL are sold, on average, for 4532.00.

O2) The average actual sales price for a bike category. For example: In 2017, the average actual sales price of racing bikes was 2321.00.

O3) The average regular sales price of bike models for a bike category. For example: In 2017, the average regular sales price of bike models of category race bikes was 2834.00.

O4) The revenue, i.e., sum of actual sales price, per bicycle model. For example: In 2017, the Challenger A2-XL generated a revenue of 78,232.00.

O5) The top-seller, i.e., the bike model with the highest revenue, per bicycle category. For example: Challenger A2-XL is the top-seller of the racing bike category.

11.2 Model

This section presents the complete solution to the challenge and is split up into sections that correspond to the ontological levels in the model. As mentioned above, there is only one optional style used in this solution, the classic potency style. This strengthens the characterization potency style in the default style.

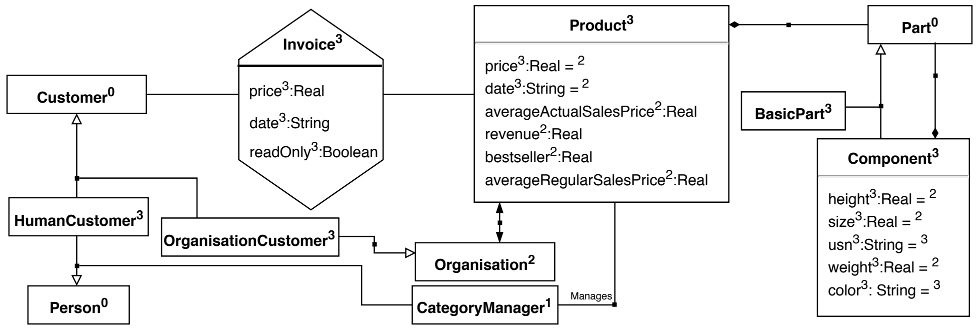


FIGURE 11.1: Bicycle Challenge first level

11.2.1 Pan-Level Constraints and Types

The model features one pan-level type, an enumeration type called *CyclistSize* to account for different predefined categories of sizes of bicycles. The definition, which is textual, has the following form: CyclistSize = [TALLCYCLIST, MEDIUMCYCLIST, SMALLCYCLIST]. The constraint for the one optional style selected, the classic potency style is Constraint 9.12.

11.2.2 Top Level (O_0): Product Types

Figure 11.1 shows the top, most abstract, ontological level of the deep model which captures the domain of selling products to customers. A *Product* is defined as a composite of *Components* and *BasicParts*. *Products* can be certified by *Organizations*. An important feature of the deep model is that this top ontological level is completely independent of the bicycle shop domain and thus can be instantiated for other domains.

11.2.3 Second Level (O_1): Bicycle Categories

Figure 11.2 shows the second level of the deep model, O_1 where the ontological instances of clabjects in O_0 reside. This level describes the structures of the different kinds of bicycle product categories as well as their different roles and stakeholders. The strict modeling style requires all ontological instances of O_0 to reside at O_1 but does not require all elements in O_1 to have a direct ontological type. For example, the clabject *ProfessionaleRacingFrame* has no ontological type. This is because it needs more attributes than a "normal" *Component*, so it inherits the normal component attributes from *Frame* (an instance of *Component*) and adds its own attributes relevant to professional racing frames. There are two connections needed between *BicycleConfiguration* and *Wheel*, one for the front wheel and one for the rear wheel. Every

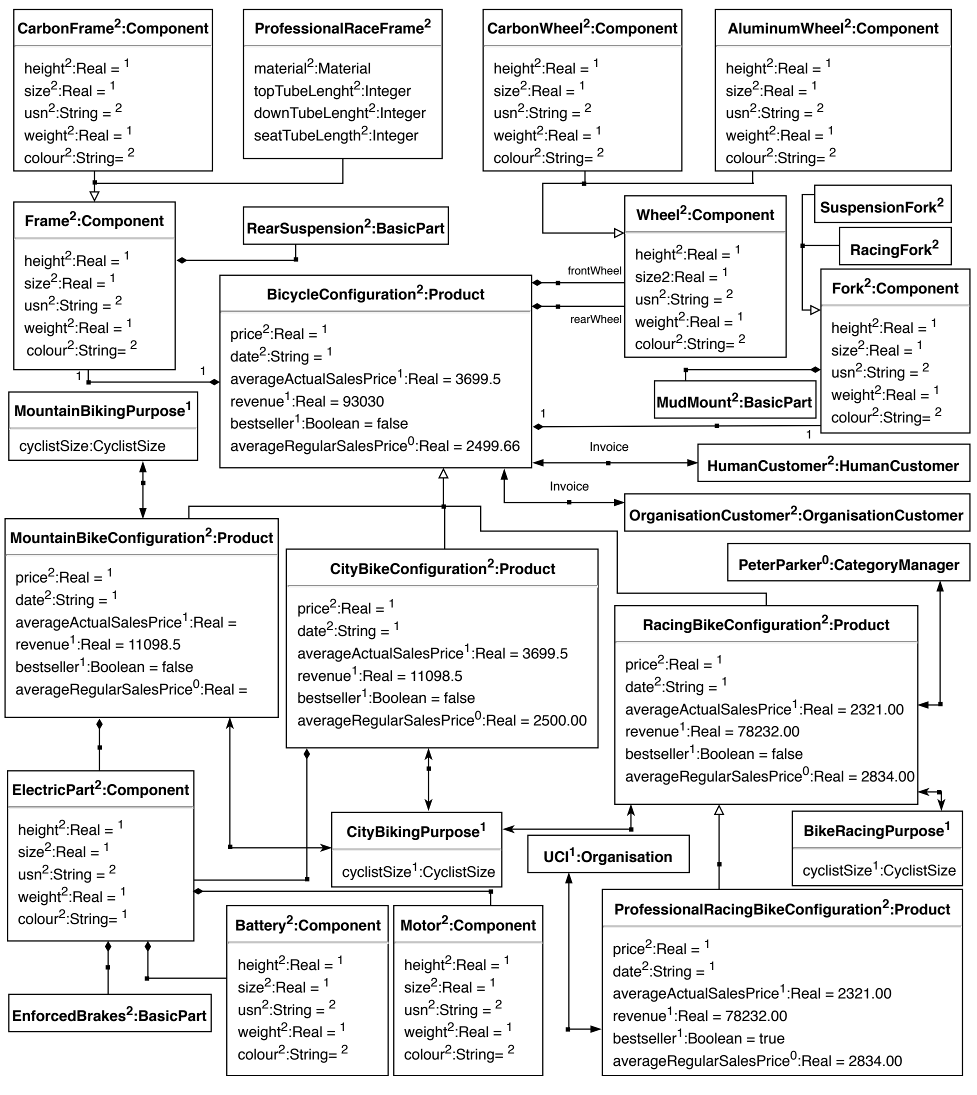


FIGURE 11.2: Bicycle Challenge second level

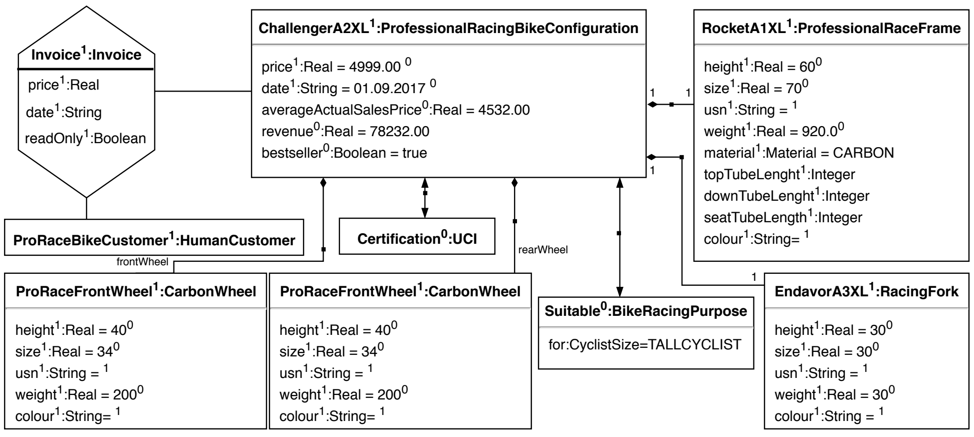


FIGURE 11.3: Bicycle Challenge third level

instance of *Product* is connected to a *Purpose*. Note that all the connections are depicted in the collapsed form in Figure 11.2. Although this means their attributes cannot be seen, it is still possible to display their names in the style of UML associations. Multiple DOCL constraints are defined at this level to ensure its compliance with the requirements, Constraints 11.1, 11.2, 11.3, 11.4, 11.5, 11.6, 11.7, 11.8. These are explained in Section 11.3.

11.2.4 Third Level (O_2): Bicycle Configurations

Figure 11.3 shows the example bicycle configuration described in the Bicycle Challenge. This configuration, called *ChallengerA2XL* is an instance of the *ProfessionalRacingBike* category and has a regular sales price of 4999.00. The *averageActualSalesPrice* is determined by the derive Constraint 11.9 to have a value of 4349.25. The derived value for the *revenue* attribute (for 2017) is 78232.0 as described in the Challenge description. It is also the best-selling configuration at this level. The connected components represent the minimum configuration satisfying the constraints defined at the level above. For instance, the connected wheels both have the same size and can be distinguished by the *instanceOf* connection. The front wheel is connected to the front wheel connection instance and the rear wheel to the rear wheel connection.

11.2.5 Bottom Level (O_3): Bicycle Configurations

Figure 11.4 shows the bicycle configuration instances described in the Challenge description at the lowest (most concrete) level of the deep model. These model elements are individuals. The *Invoice* connection captures the sales

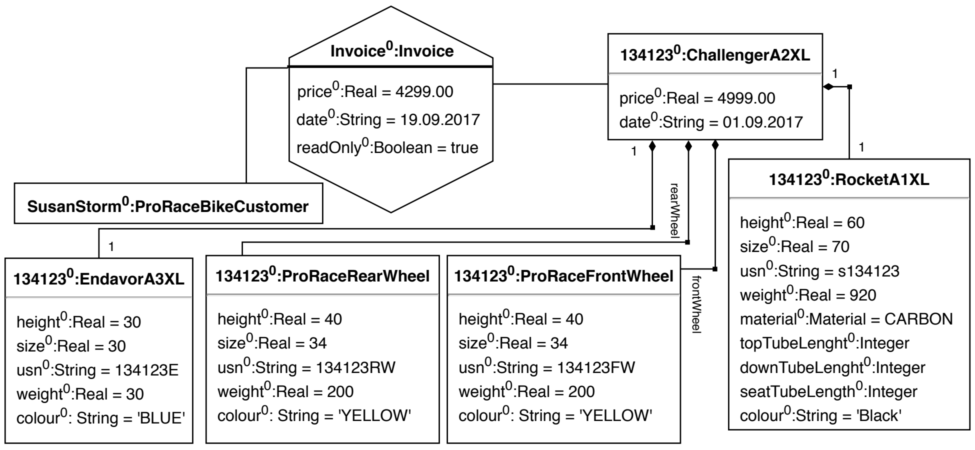


FIGURE 11.4: Bicycle Challenge fourth level

transaction and is the source of information for the derive constraints for the attributes *revenue*, *bestseller*, and *averageActualSalesPrice*. The bicycle concerned is an instance of *ChallengerA2XL* sold for a price of 4299.00 instead of the regular price of 4999.00 as shown in the *Invoice* connection. The values of the attribute of the clabjects derived from *Component* and *Product* cannot be changed at this level because their mutability is set to zero at the level above, except for the attributes *colour* and *usn* of the *Component* clabject.

11.3 Fulfillment of the Requirements

The model presented in the previous section fulfills all the requirements, as explained in this section.

M1) The model at level O_0 defines a *Product*, or bicycle configuration, as a composition of *BasicPart* and *Component*.

M2) The *Component* clabject contains attributes, like *height*, *size*, *weight*, etc.

M3) The *BicycleConfiguration* clabject at O_1 established contains relationships to all of the mandatory bicycle parts (*Wheel*, *Fork* and *Frame*). The constraint that the wheel sizes have to match is derived by the following constraint:

```
context BicycleConfiguration
inv O11-wheelSize: self.frontWheel.size = self.rearWheel.size
```

CONSTRAINT 11.1: Front and rear wheel size have to match

M4) The subclasses of *BicycleConfiguration* which are *CityBikeConfiguration*, *MountainBikeConfiguration*, *RacingBikeConfiguration*, and *RacingBikeConfiguration* are bicycles that are made for different purposes.

M5) The fulfillment of this requirements has three elements –

1. The *ElectricPart* as an instance of *Component* is connected to every subclass of *BicycleConfiguration* except the *RacingBikeConfiguration*.

2. The *MountainBikeConfiguration* and *CityBikeConfiguration* can be connected to the suspension parts.

3. The constraint listed below states that the *CityBikeConfiguration* may not have a rear suspension.

   ```
   context CityBikeConfiguration
   inv O11-rearSuspension: self.frame.rearSupension -> size() = 0
   ```

   CONSTRAINT 11.2: City bikes do not have a rear suspension

4. The constraint below restricts *RacingBikeConfiguration* to be connected only to *RacingFork* and *ProfessionalRacingFrame*

   ```
   context RacingBikeConfiguration
   inv O12-racingForkFrame: self.fork.isDeepKindOf(RacingFork) and
   self.frame.isDeepKindOf(ProfessionalRacingFrame)
   ```

   CONSTRAINT 11.3: Racing bikes have to have racing forks and frames

5. The constraint below expresses that the *RacingFork* must not contain a suspension and *RacingBikeConfiguration* is not allowed to contain a mud mount

   ```
   context RacingBikeConfiguration
   inv O13-mudMount: not (self.fork.isDeepKindOf(SupensionFork)) and
   self.mudMount -> size() = 0
   ```

   CONSTRAINT 11.4: No mud mount and no suspension for *RacingBikeConfigurations*

6. The *ProfessionalRaceFrame* has the appropriate attributes.

7. The *ProfessionalRaceFrame* contains an attribute called *material* that is an enumeration type that specifies that the attribute can be of *ALUMINUM*, *STEEL*, or *CARBON*

8. The *RacingBikeConfiguration* can be connected to the *UCI* clabject to get certification.

9. Two constraints are in place to ensure the certification by the *UCI* and that the frame is made from the appropriate material.

```
context ProfessionalRacingBikeConfiguration
inv O_1 4-certification: self.organisation -> size() >= 1
```

CONSTRAINT 11.5: Mandatory certification for *ProfessionalRacingBikeConfigurations*

```
context ProfessionalRacingBikeConfiguration
inv O_1 5-frameMaterial: self.frame.material.#getLiteral()# = "ALUMINUM" or
self.frame.material.#getLiteral()# = "CARBON"
```

CONSTRAINT 11.6: Frame material for professional racing bikes

```
context ProfessionalRacingBikeConfiguration
inv O_1 6-minimumWeight: self.frame.weight >= 5200
```

CONSTRAINT 11.7: The minimum weight of a certified racing bike

10. The constraint below enforces carbon wheels for carbon frames

```
context BicycleConfiguration
inv O_1 7-carbonFrame: self.frame.isDeepKindOf(CarbonFrame) implies
(self.frontWheel.isDeepKindOf(CarbonWheel) or
self.frontWheel.isDeepKindOf(AluminumWheel))
```

CONSTRAINT 11.8: Carbon frame implies carbon wheels of aluminum wheels

M6) The *PeterParker* clabject (at level O_1) is an instance of *CategoryManager* (at level O_0), which is a subclass of *Person* (an abstract clabject). *CategoryManager* can be connected to *Product*.

M7) The *ChallengerA2-XL* clabject at level O_2 is an instance of the *ProfessionalRacingBike* clabject and is connected to the *Suitable* clabject which is an instance of *BikeRacingPurpose*. The attribute *for* has the value *TALLCYCLIST*. The *RocketA1XL* clabject is a *ProfessionalRaceFrame* which has a weight of 920 grams.

M8) The **M7** requirement is instantiated at level O_3.

M9) At level O_0 *Product* is connected via the *Invoice* connection to *Customer*. This clabject is further specialized into *HumanCustomer* and *OrganisationCustomer*. The *Invoice* clabject contains an attribute of type Boolean which is called *readOnly*.

M10) Each bicycle model has a regular sales price. The regular sales price of Challenger A2-XL is EUR 4999.00. The actual sales price of physical instances of the bicycle model (i.e., the price given in an invoice) may be lower.

M11) At the lowest level, O_3, the clabject *134123* is sold via the *Invoice* connection to *SusanStorm*.

O1) The values of *averageActualSalesPrice* are defined by the following "derive" constraints that is applied at specifically defined levels. Constraint 11.9 derives the value of the *averageActualSalesPrice* for every instance of *Product* at O_1 and O_2. The context of the derivation is iterated over every instance of *Product* so that the value is derived for each individual instance.

```
context Product::averageActualSalesPrice:Real
derive O01: self.allInstances() -> select(c|c.#getPotency()# = 0) ->
select(c|c.Invoice.date.substring(7,10) = "2017") -> collectNested(Invoice.
    price)
-> sum() / self.allInstances() -> select(c|c.#getPotency()# = 0) -> size()
```

CONSTRAINT 11.9: The derive constraint for *averageActualSalesPrice*

O2) This requirement is solved by the requirement above (**O1**).

O3) The average regular sales price of bike models is also calculated for each bike category. For example: In 2017, the average regular sales price of bike models of category race bikes was 2834.00.

```
context Product::averageRegularSalesPrice:Real
derive O02: self.allInstances() -> select(c|c.#getPotency()# = 1)
-> select(date.substring(7,10) = "2017") -> collect(price) -> sum() /
self.allInstances() -> select(c|c.#getPotency()# = 0) -> size()
```

CONSTRAINT 11.10: The average regular sales price derived attribute

O4) The *revenue* attribute is also a derived attribute and the constraint calculates the value similarly.

```
context Product::revenue:Real
derive O03: self.allInstances() -> select(c|c.#getPotency()# = 0) ->
select(c|c.Invoice.date.substring(7,10) = "2017") -> collectNested(Invoice.
    price)->sum()
```

CONSTRAINT 11.11: The revenue derived attribute

O5) The bestseller attribute is derived by a reflective query.

```
context Product::bestseller:Boolean
derive O04: let topSeller:Boolean = false in
if Clabject -> select(c|c.isDeepKindOf(BicycleConfiguration) = true) ->
select(date.substring(7,10) = "2017") -> sortedBy(revenue) -> last() = self
then topSeller = true else topSeller = false endif
```

CONSTRAINT 11.12: The bestseller derivation

11.4 Discussion

As explained in Chapter 3.2, LML connections can be depicted in two forms – an exploded form using the hexagon symbol and an imploded form as a dot. In this particular model, the connection between *Product* and *Customer*, called *Invoice*, is depicted in expanded form in Figure 11.1 to show its two attributes, while the connection between *CategoryManager* and *Product*, called *Manages*, is shown in collapsed form because it has no attributes.

The attributes of *Invoice* document the essential properties of sales transactions. The other kind of relationship appearing in Figure 11.1 is a specialization that is depicted using the UML unfilled-triangle notation. The figure highlights the important point that the potencies of sub-clabjects need not be the same as those of their super-clabjects because potency is based on direct classification relationships (as opposed to indirect classification relationships arising from inheritance hierarchies).

The derived attributes *averageActualSalesPrice*, *averageRegularSalesPrice*, *revenue* and *bestseller* of the clabject *Product* are responsible for storing the optional sales information described in Section 2.2 of the Challenge description [2]. Constraint 11.10 defines the value of the *averageRegularSalesPrice* attribute in a similar manner as Constraint 11.9. Constraint 11.11 defines the value of the *revenue* attribute while Constraint 11.12 determines the top-seller of each category and the top-selling category. UML-style multiplicity constraints are used to indicate that a *BicycleConfiguration* must have exactly one *Frame* and

Fork. In addition, eight DOCL constraints are needed to fulfill the mandatory requirements (requirements starting with *M*) of the Challenge.

Beyond the default style, the presented solution applies the classic potency rule. In total 12 domain constraints were used to fulfill the requirements and help make the model more concise and precise.

Chapter 12

The Process Challenge

The second challenge defined by the MULTI Workshop community was the Process Challenge. The first chance to present a solution to this challenge was the MULTI 2019 workshop. After that, the EMISA journal created a special issue for extended descriptions of solutions to this challenge. The solution presented in this chapter is based on the one published in the EMISA journal [83].

The aim of this particular challenge is to showcase the ability of different MLM approaches to handle process languages. Although the challenge describes two processes to be modeled, this chapter only shows the solution for the "ACME Software Engineering (SE) Process", which only has three ontological levels. As well as the default Melanee style (see Section 8.3), it applies four optional styles – the universal isonymic instantiation style (see Section 9.2.2), the selective classification mandation style applied to connections (Constraint 9.2), and the universal root inheritance style for level O_0 (Constraint 9.11). In terms of patterns, at level O_1 the model uses the strong Odell powertype pattern (Constraint 10.13).

12.1 Requirements

The requirements for this challenge are split into two groups. The first group, whose identifiers start with *P*, defines requirements common to all process domains, while the second group, whose identifiers start with *S*, is tailored to a specific process in a specific domain (i.e., the so-called ACME SE Process).

P1) A *process type* (such as *claim handling*) is defined by the composition of one or more *task types* (*receive claim*, *assess claim*, *pay premium*) and their relations

P2) Ordering constraints between *task types* of a *process type* are established through *gateways*, which may be *sequencing*, *and-split*, *or-split*, *and-join* and *or-join*

P3) A *process type* has one *initial task type* (with which all its executions begin), and one or more *final task types* (with which all its executions end)

P4) Each *task type* is created by an *actor*, who will not necessarily perform it. For example, *Ben Boss* created the task type *assess claim*

P5) For each *task type*, one may stipulate a set of *actor types* whose instances are the only ones that may perform instances of that *task type*. For example, in the *XSure* insurance company, only a *claim handling manager* or a *financial officer* may *authorize payments*

P6) A *task type* may alternatively be assigned to a particular set of *actors* who are authorized (e.g., *John Smith* and *Paul Alter* may be the only *actors* who are allowed to *assess claims*)

P7) For each *task type* (such as *authorize payment*) one may stipulate the *artifact types* which are *used* and *produced*. For example, *assess claim* uses a *claim* and produces a *claim payment decision*

P8) *Task types* have an *expected duration* (which is not necessarily respected in particular occurrences)

P9) *Critical task types* are those whose instances are *critical tasks*; each of the latter must be performed by a *senior actor* and the artifacts they produce must be associated with a *validation task*

P10) Each *process type* may be enacted multiple times

P11) Each process comprises one or more tasks

P12) Each *task* has a *begin date* and an *end date*. (e.g., *Assessing Claim 123* has *begin date 01-Jan-19* and *end date 02-Jan-19*)

P13) *Tasks* are associated with *artifacts* used and produced, along with performing *actors*

P14) Every *artifact* used or produced in a task must instantiate one of the *artifact types* stipulated for the *task type*

P15) An actor may have more than one actor type (e.g., Senior Manager and Project Leader)

P16) Likewise, an artifact may have more than one artifact type

P17) An actor who performs a task must be authorized for that task. Typically, a class of actors is automatically authorized for certain classes of tasks

P18) Actor types may specialize other actor types in which case all the rules that apply to instances of the specialized actor type must apply to instances of the specializing actor type. For example, if a manager is allowed to perform tasks of a certain task type, so is a senior manager.

P19) All modeling elements, at all levels, must have a last updated value of type timestamp. This feature should be defined as few times as possible, ideally only once. Respective definitions are exempt from the requirement to have a last updated value

S1) A requirements analysis is performed by an analyst and produces a requirements specification

S2) A test case design is performed by a developer or test designer and produces test cases

S3) An occurrence of coding is performed by a developer and produces code. It must furthermore reference one or more programming languages employed

S4) Code must reference the programming language(s) in which it was written

S5) Coding in COBOL always produces COBOL code

S6) All COBOL code is written in COBOL

S7) Ann Smith is a developer; she is the only one allowed to perform coding in COBOL

S8) Testing is performed by a tester and produces a test report

S9) Each tested artifact must be associated with its test report

S10) Software engineering artifacts have a responsible actor and a version number. This applies to requirements specification, code, test case, test report, but also to any future types of software engineering artifacts

S11) Bob Brown is an analyst and tester. He has created all task types in this software development process

S12) The expected duration of testing is 9 days

S13) Designing test cases is a critical task that must be performed by a senior analyst. Test cases must be validated by a test design review

12.2 Model

This section presents the complete solution to the challenge and is split into sections that correspond to the ontological levels.

12.2.1 Pan-Level Constraints and Types

As mentioned above, the model applies two pan-level styles beyond the default style – the universal isonymic instantiation style and the selective classification mandation style for connections. The constraints enforcing these styles are Constraint 9.2 (selective classification mandation for connections (an application of the mandatory connection classification style)) and Constraints 9.3 and 9.4 (universal isonymic instantiation).

12.2.2 Top Level (O_0): Generic Process Metamodel

Figure 12.1 shows the top, most abstract, ontological level of the deep model which defines the generic process metamodel called for in the requirements. The most abstract concept in this model is the *Element* clabject which is the superclass of all clabjects at this level. This level therefore applies the universal root inheritance style. Constraint 12.1 has its context at the top level.

```
context Level(0,0)
inv: if self.#isLeafLevel()#
    then true
    else self.#getClabjects()# -> select(c|c.#getSupertypes()# ->
    size() = 0) -> closure(c|c.#getSubtypes()#)
    -> includesAll(self.#getClabjects()#)
    endif
```

CONSTRAINT 12.1: The Level 0_0 universal inheritance root style

The potency value of *ProcessType* is 2 so that it can be instantiated over two consecutive lower levels. *Actor* represents the human-played role responsible for defining instances of a particular kind of *Element*, *TaskType*, at the O_1 level

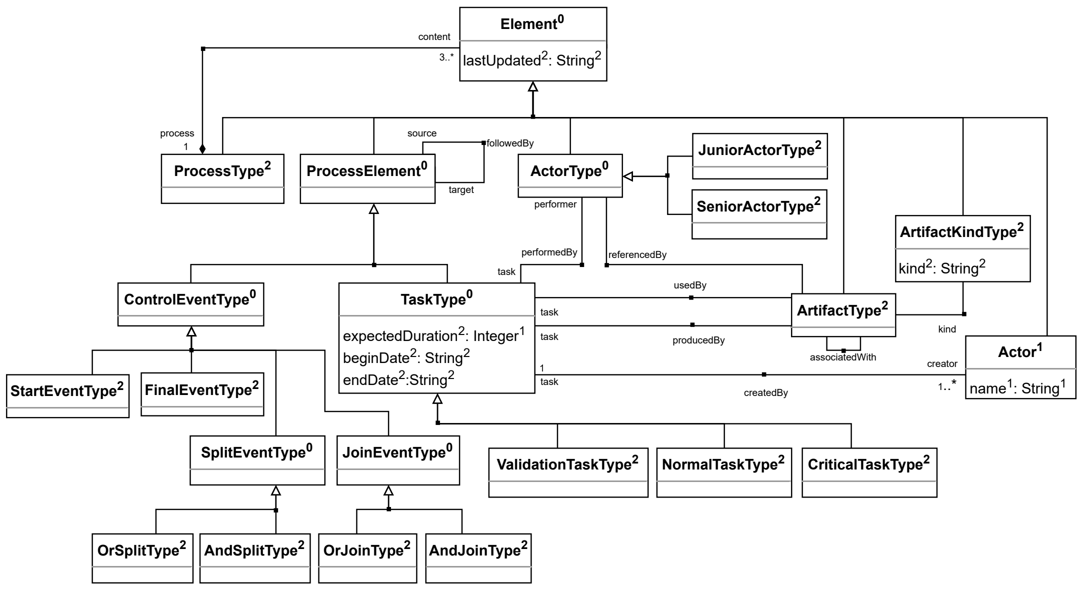
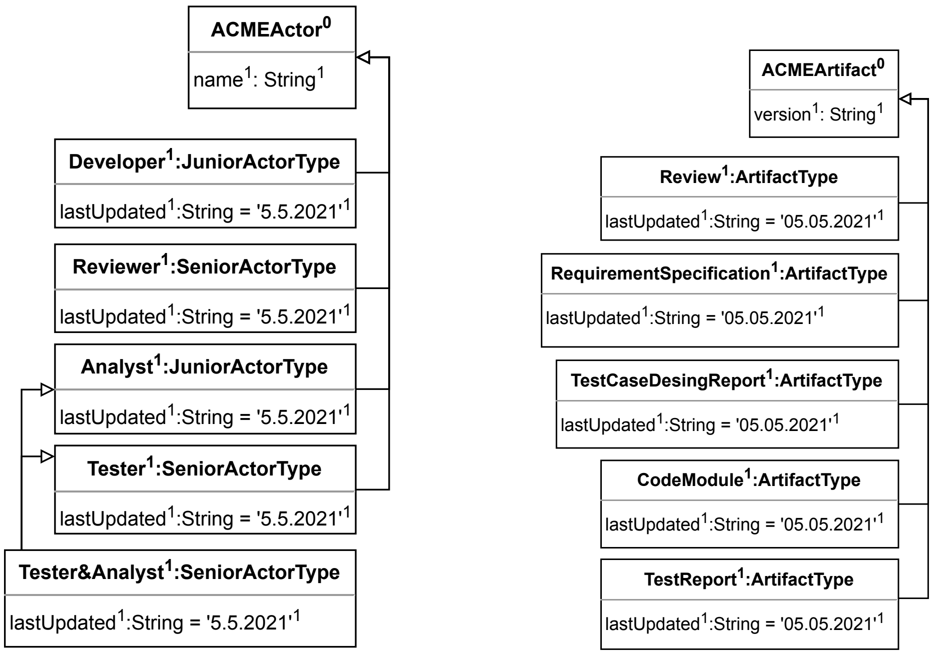


FIGURE 12.1: Process Challenge Level O_0

below. Since instances of *Actor* are therefore only needed at the O_1 level, the potency of *Actor* is 1. The *Actor* clabject is able to create tasks through the *createdBy* relationship on the level directly below.

There are five basic kinds of elements that can be contained in a process type, which are modeled as subclasses of *Element* – *ControlEventType*, *ActorType*, *TaskType*, *ArtefactType* and *ArtefactKindType*. The first three are abstract classes and therefore have a potency of 0 while the last two are concrete classes with a potency of 2. *ControlEventType* has four subclasses, two of which are concrete. *StartEventType* and *FinalEventType* have a potency value of 2 and the other two are abstract superclasses, i.e., *SplitEventType* and *JoinEventType* with a potency of 0. The four concrete subclasses of the split and join events are *AndJoin*, *AndSplit*, *OrJoin* and *OrSplit*. The *StartEventType* clabject has to be present in every instance of a process type exactly once and the *FinalEventType* has to be present at least once. The split and join events can participate in the *followedBy* relationship that is defined on *ProcessElement* in combination with any *TaskType*.

ActorTypes is specialized by two classes, i.e., *JuniorActorType* and *SeniorActorType*, while *TaskType* has three subclasses – *NormalTaskType*, *CriticalTaskType* and *ValidationTaskType*. All of the specialized clabjects have a potency value of 2. *TaskType* itself is impotent and contains three attributes which are *expectedDuration* of type integer, *beginDate* of type string, and *endDate* of type string. The durability value of all three attributes is 2, so they are present in all instances of subclasses of this clabject. They also have mutability values of 2 with the exception of *expectedDuration*. This attribute has a mutability


ACMEActor^0
name^1: String^1
Developer^1:JuniorActorType
lastUpdated^1:String = '5.5.2021'^{1}
Reviewer^1:SeniorActorType
lastUpdated^1:String = '5.5.2021'^{1}
Analyst^1:JuniorActorType
lastUpdated^1:String = '5.5.2021'^{1}
Tester^1:SeniorActorType
lastUpdated^1:String = '5.5.2021'^{1}
Tester&Analyst^1:SeniorActorType
lastUpdated^1:String = '5.5.2021'^{1}
ACMEArtifact^0
version^1: String^1
Review^1:ArtifactType
lastUpdated^1:String = '05.05.2021'^{1}
RequirementSpecification^1:ArtifactType
lastUpdated^1:String = '05.05.2021'^{1}
TestCaseDesingReport^1:ArtifactType
lastUpdated^1:String = '05.05.2021'^{1}
CodeModule^1:ArtifactType
lastUpdated^1:String = '05.05.2021'^{1}
TestReport^1:ArtifactType
lastUpdated^1:String = '05.05.2021'^{1}

(A) Actor powertype (B) Artifact powertype

FIGURE 12.2: Strong Odell Powertype Patterns

value of 1 which means its value can be changed at the level immediately below but not at the levels below that. Every task can produce artifacts that can be of any kind. This is represented by the clabjects *ArtifactType* and *ArtifactKindType* which are connected to *TaskType* by the *producedBy* and *usedBy* connections. It is the *kind* relationship that connects the artifact to the specification of what kind of artifact it is.

12.2.3 Second Level (O_1): ACME SE Process

The second level of the deep model describes the ACME Software Engineering (SE) Process as an instance of the O_0 level general process modeling language. Although the O_1 is a single, coherent model, for clarity we show it using two separate views in Figures 12.2 and 12.3. Figure 12.2 highlights the fact that the strong Odell powertype pattern is used at this level, while Figure 12.3 highlights the flow of task types in the process as defined by the requirements. The majority of clabjects appearing in the former also appear in the latter, but in both cases, classes with the same name represent the same model element. This follows the well-established UML convention that identically-named model elements, with the same direct type, represent the same model element. It would be possible to show the information in Figures 12.2 and 12.3 in one figure, but this would be much more cluttered.

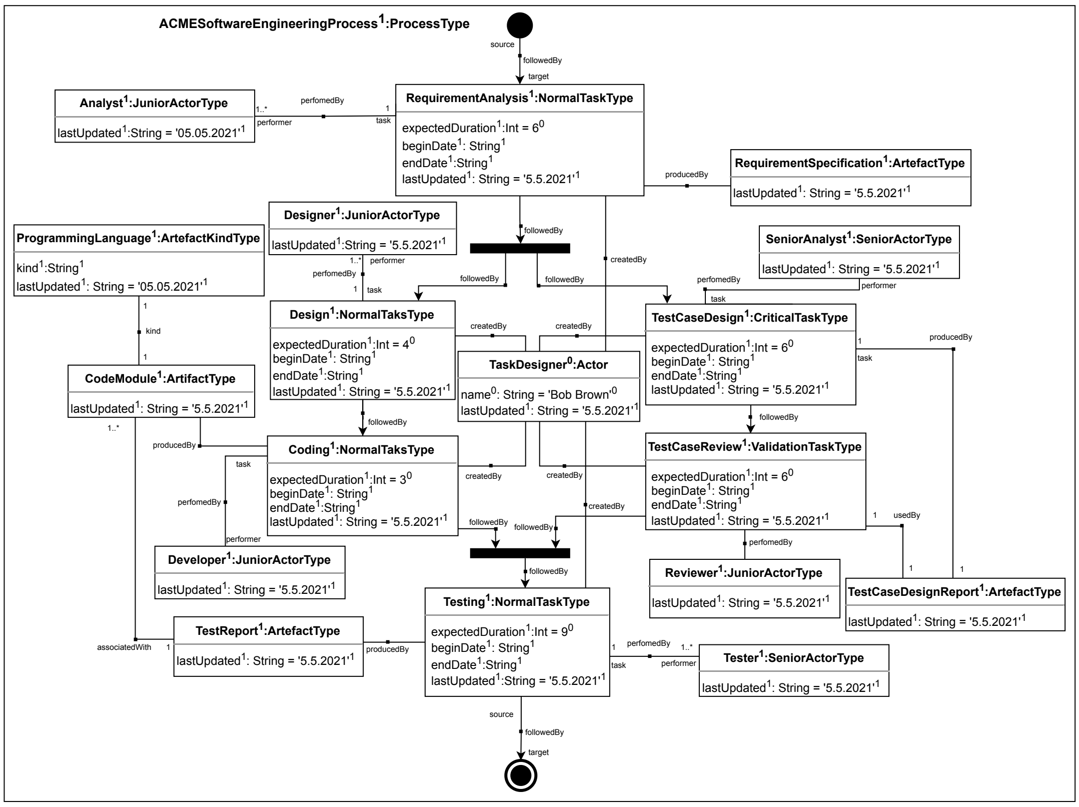

ACMESoftwareEngineeringProcess¹:ProcessType
RequirementAnalysis¹:NormalTaskType
Analyst¹:JuniorActorType
RequirementSpecification¹:ArtefactType
Designer¹:JuniorActorType
ProgrammingLanguage¹:ArtefactKindType
SeniorAnalyst¹:SeniorActorType
Design¹:NormalTaksType
TaskDesigner⁰:Actor
TestCaseDesign¹:CriticalTaskType
CodeModule¹:ArtifactType
Coding¹:NormalTaksType
TestCaseReview¹:ValidationTaskType
Developer¹:JuniorActorType
Reviewer¹:JuniorActorType
TestCaseDesignReport¹:ArtefactType
Testing¹:NormalTaskType
TestReport¹:ArtefactType
Tester¹:SeniorActorType
followedBy
perfomedBy
producedBy
createdBy
usedBy
associatedWith
kind
performer
task
source
target

FIGURE 12.3: Task Type Flow in the ACME SE Process

Figure 12.2 focuses on the specific *ArtifactTypes* and *ActorTypes* comprising the ACME SE Process and describes their relationships in terms of generalization sets. The generalization set for *ACMEActor* shows that there are five specific *ActorTypes* in the process, *Developer*, *Reviewer*, *Analyst*, *Tester* and *Tester&Analyst* and that the latter is a specialization of both *Analyst* and *Tester*. This means that an instance of *Tester&Analyst* can serve as (i.e., play the role of) an instance of *Analyst* or *Tester*. The generalization set for *ACMEArtifact*, which shows that there are five specific *Artifact Types* in the process, *Review*, *RequirementSpecification*, *TestCaseDesignReport*, *CodeModule* and *TestReport*, plays two important roles.

To ensure that any *ActorType* added in the future have the same properties as the current *ActorTypes* represented in the model, the strong Odell powertype pattern defined in Section 10.3.2 is applied to *ACMEActor*. More specifically, *ActorType* is defined to be the string characterizer of *ACMEActor*. The concrete applications of the generic Constraint 10.13 to *ActorType* and *ACMEActor* are shown below in Section 12.3.

Figure 12.3 focuses on describing the structure of the ACME SE Process in terms of the ordering constraints between the specific *Task Types* appearing in the process, as well as their relationship to all specific *Actor Types* and *Artifact*

Types shown in Figure 12.2. It also shows the specific *Actor* responsible for their design. The process starts with the *RequirementsAnaysis* task type and then splits up the enactment of the process with an instance of *AndSplit*. The left-hand side of the separated flow takes care of the *Design* and the *Coding* tasks of the process. One or more instances of *Designer* perform the *Design* task and *Coding* is performed by one or more *Developers*. The right-hand side introduces the *TestCaseDesign* as a *CriticalTaskType* which is performed by a *SeniorAnalyst* which in turn is an instance of *SeniorActorType*. It also produces a *TestCaseDesignReport* which is used in the *TestCaseReview* task which follows the *TestCaseDesign* as an instance of *ValidationTaskType*. This task is performed by a *Reviewer*. The *AndJoin* that re-connects the flow of the process is followed by the *Testing* task which produces a *TestReport* artifact and is performed by a *Tester*. The *TestReport* artifact is then associated with another artifact which is *CodeModule*. Due to the fact that the *Coding* task can produce multiple *CodeModules*, the multiplicity constraint on the *associatedWith* connection is 1 on the *TestReport* end and '1..*' on the *CodeModule* end. Every *CodeModule* has to reference the programming language it is written in which is captured by the *ArtifactKindType* instance, *ProgrammingLanguage*, with the *kind* attribute. All instances of *TaskType* are created by the *TaskDesigner* whose name is "Bob Brown". This is an instance of *Actor* at level O_0.

12.2.4 Third Level (O_2): An ACME SE Process Enactment

Figure 12.4 shows an example enactment of the ACME SE process at the O_2 level to generate a software system called Simple System. Since the ACME SE process has no or-splits nor or-joins, the basic content and layout of the model are similar to the O_1 level. Basically, every task type in the O_1 level has an instance at the O_2 level with the corresponding *followedBy* connection. The main difference is that this enactment model identifies the actual indirect instances of *ACMEActor* that carried out each task. For example, *Ann Smith* performed the *Coding* task called *SSCoding*. This also compels every instance to have a *name* attribute. Every indirect *ACMEArtifact* instance also has to have a *version* attribute due to the power type pattern enforced at the level above (O_1).

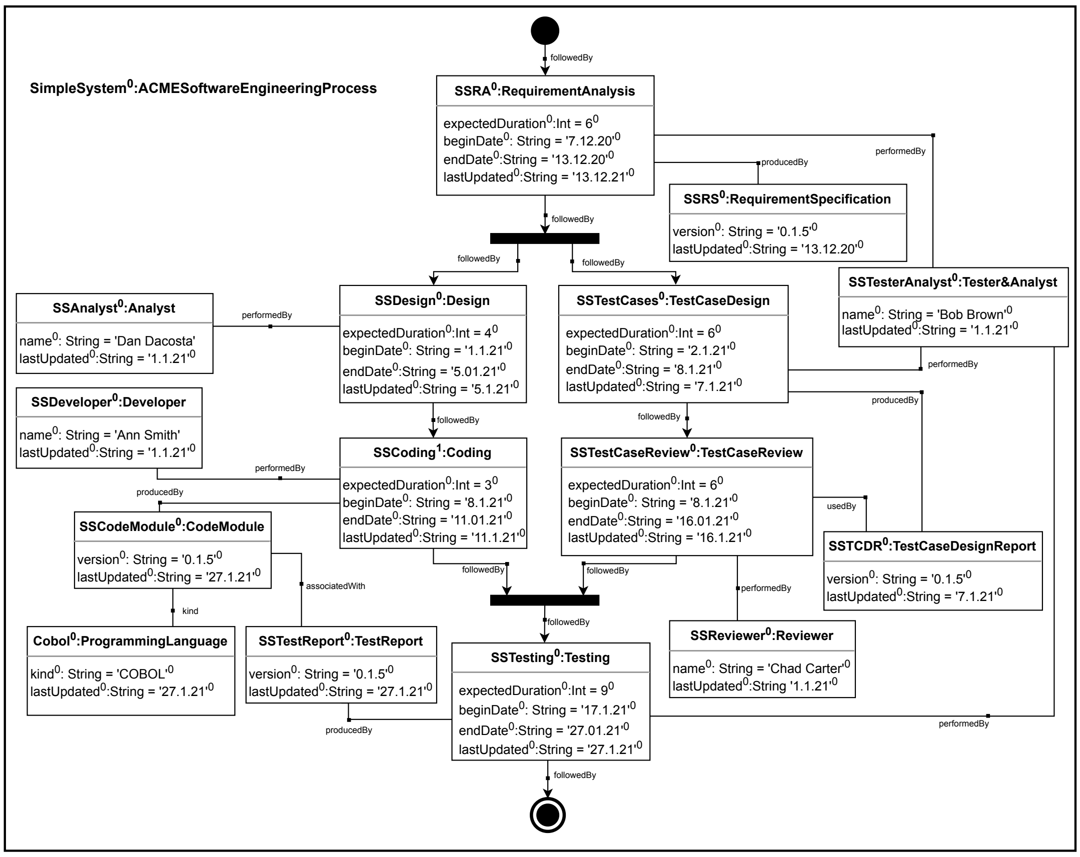

SimpleSystem⁰:ACMESoftwareEngineeringProcess
followedBy
SSRA⁰:RequirementAnalysis
expectedDuration⁰:Int = 6⁰
beginDate⁰: String = '7.12.20'⁰
endDate⁰:String = '13.12.20'⁰
lastUpdated⁰:String = '13.12.21'⁰
performedBy
producedBy
SSRS⁰:RequirementSpecification
version⁰: String = '0.1.5'⁰
lastUpdated⁰:String = '13.12.20'⁰
followedBy
followedBy
followedBy
SSTesterAnalyst⁰:Tester&Analyst
name⁰: String = 'Bob Brown'⁰
lastUpdated⁰:String = '1.1.21'⁰
SSAnalyst⁰:Analyst
name⁰: String = 'Dan Dacosta'
lastUpdated⁰:String = '1.1.21'⁰
performedBy
SSDesign⁰:Design
expectedDuration⁰:Int = 4⁰
beginDate⁰: String = '1.1.21'⁰
endDate⁰:String = '5.01.21'⁰
lastUpdated⁰:String = '5.1.21'⁰
SSTestCases⁰:TestCaseDesign
expectedDuration⁰:Int = 6⁰
beginDate⁰: String = '2.1.21'⁰
endDate⁰:String = '8.1.21'⁰
lastUpdated⁰:String = '7.1.21'⁰
performedBy
producedBy
SSDeveloper⁰:Developer
name⁰: String = 'Ann Smith'
lastUpdated⁰:String = '1.1.21'⁰
followedBy
followedBy
SSCoding¹:Coding
expectedDuration⁰:Int = 3⁰
beginDate⁰: String = '8.1.21'⁰
endDate⁰:String = '11.01.21'⁰
lastUpdated⁰:String = '11.1.21'⁰
performedBy
producedBy
SSTestCaseReview⁰:TestCaseReview
expectedDuration⁰:Int = 6⁰
beginDate⁰: String = '8.1.21'⁰
endDate⁰:String = '16.01.21'⁰
lastUpdated⁰:String = '16.1.21'⁰
usedBy
SSCodeModule⁰:CodeModule
version⁰: String = '0.1.5'⁰
lastUpdated⁰:String = '27.1.21'⁰
SSTCDR⁰:TestCaseDesignReport
version⁰: String = '0.1.5'⁰
lastUpdated⁰:String = '7.1.21'⁰
associatedWith
followedBy
followedBy
performedBy
kind
followedBy
Cobol⁰:ProgrammingLanguage
kind⁰: String = 'COBOL'⁰
lastUpdated⁰:String = '27.1.21'⁰
SSTestReport⁰:TestReport
version⁰: String = '0.1.5'⁰
lastUpdated⁰:String = '27.1.21'⁰
SSTesting⁰:Testing
expectedDuration⁰:Int = 9⁰
beginDate⁰: String = '17.1.21'⁰
endDate⁰:String = '27.01.21'⁰
lastUpdated⁰:String = '27.1.21'⁰
SSReviewer⁰:Reviewer
name⁰: String = 'Chad Carter'⁰
lastUpdated⁰:String '1.1.21'⁰
producedBy
performedBy
followedBy

FIGURE 12.4: Process Challenge Level O_2

12.3 Fulfillment of the Requirements

This subsection describes how each of the requirements, defined in the Challenge, are satisfied by the O_0 level of our solution, with suitable DOCL constraints being introduced where necessary.

Fulfillment of the General Process Model Requirements

P1) This is supported by the composition relationship between *Element* and *ProcessType*. Every instance of *Element* is contained in one instance of *ProcessType*, and every instance of *ProcessType* contains at least three *Elements*. The following constraint ensures that at least one of these contained elements is a *TaskType*.

```
context ProcessType(1,2)
inv: self.content -> exists(element|element.deepOCLTypeOf(TaskType))
```

P2) The sequencing relationships between task types are achieved by means of the *followedBy* relationship between *ProcessElements* while the split and join gateways are realized by dedicated clabjects which are subclasses of *ControlEventType*.

Instances of *TaskType* must be involved in two *followedBy* connections. One connection, in which it participates as the target, is to the *ProcessElement* that precedes it, and the other, in which it participates as the source, is to the *ProcessElement* that follows it.

```
context TaskType(1,2)
inv taskFollowedBy: self.source -> size() = 1 and self.target -> size() = 1
```

The instances of *StartEventType* are not allowed to participate in a *followedBy* relationship as the target, and must therefore be the beginning of a process. It must reach exactly one element through the *followedBy* connection, where it is the source of the connection.

```
context StartEventType(1,2)
inv start: self.source -> size() = 0 and self.target -> size() = 1
```

The instances of *FinalEventType* are not allowed to have a follower, and must therefore be the end of a process.

```
context EndEventType(1,2)
inv end: self.source -> size() = 1 and self.target -> size() = 0
```

No instance of *SplitEventType* is allowed to have only one incoming connection but has to have at least two outgoing connections.

```
context SplitEventType(1,2)
inv split: self.source -> size() = 1 and self.target -> size() => 2
```

For the instances of *JoinEventType* the rule on how many outgoing and incoming connections they can have is exactly the reverse of split events.

```
context JoinEventType(1,2)
inv join: self.source -> size() => 2  and self.target -> size() = 1
```

P3) To simplify the definition of these well-formedness rules, we avoid the introduction of dedicated clabjects for initial task types and final task types since these would also need to be distinguished as being normal, critical, and validation tasks, and instead we identify initial and final tasks by their connection to start and finish control event types respectively (i.e., *StartEventType* and *FinalEventType*). A task type is, therefore, an initial task type if it is the target in a *followedBy* connection with an instance of *StartEventType*, and is a final task type if it is the source in a *followedBy* connection with an instance of *FinalEventType*. The following constraint ensures that a *ProcessType* has the correct number of *StartEventTypes* and *FinalEventTypes*.

```
context ProcessType(1,2)
inv: self.content -> one(element|element.deepOCLTypeOf(StartEventType)) and
     self.content -> exists(element|element.deepOCLTypeOf(FinalEventType))
```

P4) This requirement is supported by the mandatory *createdBy* relationship between *TaskType* and *Actor* which has a multiplicity constraint with a lower bound of 1 at the *Actor* end.

P5) This feature is enabled by the *performedBy* relationship between *TaskType* and *ActorType* which has 0..* multiplicity and thus is optional. The specific authorizations that are applicable in a particular scenario are established by the instances of these clabjects.

P6) Our approach supports this capability by allowing constraints to be defined at the O_1 level that control which instances of specific *ActorTypes* can enter into *performedBy* relationships with specific *TaskTypes* at the O_2 level. Such constraints therefore effectively declare which individuals (identified by their names) are authorized to perform which tasks. The constraint used to meet requirement S7 is an example.

P7) This requirement is supported by the *usedBy* and *producedBy* relationships between *TaskType* and *ArtifactType*.

P8) This is modeled by the *expectedDuration* feature which all offspring of *TaskType* receive by virtue of the fact that it has durability 2. The mutability of the *expectedDuration* is set to 1 so that specific instances of *TaskType* for a particular scenario at the O_1 level can change it to the appropriate value for that *TaskType*. However, because the mutability of the *expectedDuration* attributes at O_1 then become 0, instances of a specific *TaskType* at the O_2 level cannot assign a new value to *expectedDuration*, they must retain the value set at O_1. The *expectedDuration* of *TaskType* at the O_0 level is set to "undefined".

P9) The concepts of critical task types and senior actor types are modeled by the *CriticalTaskType* and *SeniorActorType* sub-clabjects of *TaskType* and *ActorType*, respectively. The fact that critical task types can only be performed by senior actor types is captured by the following constraint on the *performedBy* relationship between task types and actor types.

```
context CriticalTaskType(1,2)
inv: self.performer -> forAll(p|p.deepOCLKindOf(SeniorActorType)) and self.
     target.isDeepOCLTypeOf(ValidationTaskType)
```

P10) This is supported by the basic mechanics of the deep modeling approach which allows the instances of the O_0 level metamodel, at O_1, to be instantiated again at O_2. The enactment of a process type is captured by the enactment of the specific task types it contains, which in turn is captured by their instantiation at the O_2 level.

P11) The containment relationship multiplicities define that an instance of *ProcessType* must contain at least one indirect instance of *Element*. But in order to ensure that at least one indirect instance of *TaskType* is present in the containment the following DOCL constraint is needed.

```
context ProcessType(1,2)
inv: self.content -> select(c|c.isDeepKindOf(TaskType)) -> size() > 0
```

P12) This requirement is realized by means of the *beginDate* and *endDate* attributes of *TaskType* which capture the start and end time of *TaskType* instances and their instances, in turn. The durability and mutability values of these attributes are set to 2 so that their values can be changed at any ontological level.

P13) This requirement is supported by the *usedBy* and *producedBy* connections between *TaskType* and *ArtifactType* as well as the *producedBy* connection between *TaskType* and *ActorType*

P14) The ability to stipulate artifact types used and produced by a specific task type is supported by the *usedBy* and *peformedBy* relationships mentioned above. The actual stipulation for a particular scenario takes place at the O_1 level.

P15) The ability to declare that there are specific actor types that can perform multiple roles is supported by the multiple inheritance capability at the O_1 level.

P16) The ability to declare that a specific artifact can be regarded as being instances of multiple artifact types is supported by the multiple inheritance capability at the O_1 level.

P17) The strategy for supporting authorization is described in the discussion for P6 above. This requirement does not stipulate at which point (i.e., in which characterization context) the authorization takes place. The authorization is, therefore, declared at process enactment time (i.e., in the process enactment characterization content) and uses an actor's *instanceOf* relationship to an actor type to designate authorization to perform instances of the task type performed by that actor type.

P18) This is naturally supported by the use of the specialization relationship at the O_1 level.

P19) The existence of this value for modeling elements is captured by the *lastUpdated* String attribute of the *Element* class at the root of the inheritance hierarchy, which has durability and mutability of 2. This, in turn, means every model element in the deep model has this attribute and can set it to any value. The actual mechanism for arranging for this attribute to obtain the correct value, automatically, when an update event occurs is the *oclGetCurrentDate* operation of DOCL which returns the current date as a *String*. Melanee can be configured to trigger such an update whenever a model element is edited which automatically sets the correct timestamp as the value of this attribute.

Fulfillment of the ACME Specific Requirements

The authors of the challenge formulated 13 domain-specific requirements for the ACME software engineering process. However, one requirement, S2, is explicitly flagged as being overridden by another requirement, S13, leaving only 12 active requirements. In this section, we explain how these requirements have been fulfilled and introduce the required DOCL constraints where necessary.

In order to enforce the powertype pattern for any future instances of *ArtifactType* and *ActorType* we have made use of the DOCL capability to define constraints on any level of a deep model. Constraint *powerArtifact* ensures that every instance of *ArtifactType* at the level O_1 of the ACME SE Process application has to be connected via an inheritance relationship to *ACMEArtifact* as a subclass. In combination with the invariant constraint called *powerACMEArtifact*, where *ACMEArtifact* is used as the context to make sure that every subclass has to be of type *ArtifactType*, this prevents any untyped or wrongly typed clabject participating as a subclass in this generalization set.

```
context AtrifactType(0,0)
inv: let inheritace = self.#getDirectInstances()# -> first().#
    getInheritanceAsSubtype()# in
        inheritance.#supertype#.#supertype# -> size() = 1
        let baseType = inheritance.#supertype#.#supertype# -> first() in
        self.#getDirectInstances()# -> forAll(inst|inst.#
    getInheritanceAsSubtype()# = inheritance) and
        baseType.#getSubtypes()# -> forAll(sub|sub.isInstanceOf(self))
```

CONSTRAINT 12.2: Strong Odell Powertype pattern for the *Artifact* generalization set

The invariant constraints *powerActor* and *powerACMEActor* deal with the same problem as the constraints introduced for the *ActorType* power type pattern but instead of using the *oclIsTypeOf* operation, these constraints use the *oclIsKindOf* operation since *ActorType* is an impotent clabject and is not able to produce direct offspring.

```
context ActorType(0,0)
inv: let inheritace = self.#getDirectInstances()# -> first().#
    getInheritanceAsSubtype()# in
        inheritance.#supertype#.#supertype# -> size() = 1
        let baseType = inheritance.#supertype#.#supertype# -> first() in
        self.#getDirectInstances()# -> forAll(inst|inst.#
    getInheritanceAsSubtype()# = inheritance) and
        baseType.#getSubtypes()# -> forAll(sub|sub.isInstanceOf(self))
```

CONSTRAINT 12.3: Strong Odell Powertype pattern for the *Actor* generalization set

S1) This requirement is captured by the *performedBy* connection between *RequirementsAnalysis* and *Analyst* as well as the *producedBy* relationship between *RequirementsAnalysis* and *RequirementsSpecification*. The multiplicity constraint on the connection ensures that a *RequirementAnalysis* is only performed by one or more *Analysts*.

S2) Overridden.

S3) The first part of this requirement is fulfilled by the *performedBy* connection between *Coding* and *Developer* as well as the *producedBy* relationship between *CodeModule* and *Coding*. The second part is fulfilled by the *kind* relationship between *CodeModule* and *ProgrammingLanguage* with a multiplicity constraint of 1 on both connection ends.

S4) This is fulfilled by the *kind* connection between *CodeModule* and *ProgrammingLanguage*.

S5) This requirement only makes sense for a coding task that only involves one programming language. If the *CodeModule* is connected to a *ProgrammingLanguage* via the *kind* relationship and it references COBOL then the language of the *CodeModule* is COBOL.

S6) This requirement is again fulfilled by the *kind* connection between *CodeModule* and *ProgrammingLanguage*. By reifying the notion of programming language, and representing the language in which a programming language is written by means of a connection to a programming language object rather than by a String attribute, this requirement is automatically fulfilled by the creation of the connection.

S7) The first part of this constraint says that *Ann Smith* is a *Developer* which means that at level O_2 an instance of *Developer* has to exist with the name *Ann Smith*. This is captured by the following constraint

```
context Developer(1,1)
inv AnnSmith: self.allInstances() -> exists(d|d.name = 'Ann Smith')
```

The second part of this statement says that if a coding task exists that uses 'COBOL' as the programming language the only actor that can perform this task is *Ann Smith* which has to be a *Developer*. The fact that *Ann Smith*, as a *Devloper*, can only be connected with tasks that are instances of *Coding* tasks is ensured by the *performedBy* connection.

```
context ProgrammingLanguage (2,2)
inv: self.kind = 'COBOL' implies self.CodingModule.Coding.performer = 'Ann
    Smith'
```

The constraint defined in *ProgrammingLanguage* is only evaluated at level O_2.

S8) This requirement is again fulfilled by the *performedBy* connection between *Testing* and *Tester* and between *Testing* and *TestReport*.

S9) This requirement is fulfilled by the *associatedWith* connection between *CodeModule* and *TestReport*.

S10) The first part of this requirement is fulfilled by assigning actors to the respective tasks they can perform. Due to the general typing restrictions on connections, it is impossible to introduce new connections that connect a *Developer* to *TestCaseDesign*, for example. From the produced artifact that is connected to the task that, in turn, is connected to the actor that performs the task, we can determine the responsible actor implicitly.

The second part of this requirement is fulfilled by including the generalization sets for *ACMEActor* and *ACMEArtifact* and using these to indicate, through inheritance, that all specific *ArtifactTypes* have a version attribute and all *ACMEActors* have a name.

The third part of this requirement, stipulating that new artifact types added to future versions of the ACME SE Process must also satisfy the first part of the requirement, is fulfilled by declaring the clabject *TaskType* at the O_0 level to be the powertype of *ACMEArtifact* at the O_1 level. This ensures that every future instance of *TaskType* is a subclass of *ACMEArtifact* and thus has the required version attribute and connection to *ACMEActor*. The constraints that enforce these powertype patterns were defined in Constraints 12.2 and 12.3.

S11) The first part of this requirement is fulfilled by the following constraint.

```
context Tester&Analyst(1,1)
inv BobBrown: self.allInstances() -> exists(d|d.name = 'Bob Brown')
```

The second part is fulfilled by the *createdBy* relationship between *TaskDesigner* and all the instances of *TaskType* in the model.

S12) This requirement is fulfilled by the fact that the value of the attribute *expectedDuration* in *Testing* at level O_1 is set to 9 (which means 9 days) and the fact that the vitality property Mutability is set to the value 0. This means that the *expectedDuration* attribute of instances of *Testing* must have that same value.

S13) This requirement is captured by the fact that the clabject *TestCaseDesign* is an instance of *CriticalTaskType*, and by the *performedBy* connection between *TestCaseDesign* and *SeniorAnalyst* which is an instance of *SeniorActorType*.

12.4 Discussion

Overall, the solution to this challenge applied four optional styles, in addition to the core style, and two patterns. Two of the optional styles apply to the whole deep model (i.e., to all ontological levels) by adding the respective constraints in the pan-level part of the model. The selective classification mandation style was applied to restrain the types of connections that can be used in the lower levels to those defined at level O_0. All connections in level O_1 and O_2 have to be offspring of the connections defined in O_0. The universal isonymic instantiation style was applied to ensure that modelers could not add further features (i.e., attributes and methods) to all ontologically typed clabjects by applying the universal isonymic instantiation style.

In addition, two styles were applied at specific levels. The universal root inheritance style was applied at level O_1, while the strong Odell powertype pattern was applied at level O_2.

In terms of patterns, the presented solution applies one pattern - the strong Odell powertype pattern. The former is used at the O_0 to enable the arbitrary nesting of process elements, while the latter is applied in order to prevent the user from falsely characterizing new types of *Actors* and *Artifacts*.

Chapter 13

The Collaboration Challenge

The third challenge defined by the MULTI Workshop community was the so-called MULTI 2022 Collaboration Challenge to encourage different groups to compare and contrast their approaches in a single paper. The focus of the challenges was on describing the relationships between companies, factories, and the products they own and manufacture, respectively. The first LML solution to this challenge was published at MULTI 2021 when it was compared to DLMA [21], however, in this chapter we present an updated version that was published in [79] when it was compared to DLM.

Like the solution to the previous challenge, the solution to the Collaboration Challenge only has three ontological levels. Moreover, no optional styles are deployed beyond the default Melanee style (see Section 8.3). The solution does not employ any of the patterns defined in Chapter 10 either.

13.1 Requirements

This challenge defines thirteen requirements.

P1) A company has (a) a name, (b) owns factories, (c) owns device models

P2) Huawei is a (a) company that (b) owns Factory124 and (c) owns mobile phone models S400 and S500

P3) A factory (a) produces devices, (b) supports a list of device models, (c) can only produce devices that conform to (are of) supported device models

P4) A device conforms to a device model

P5) A device model captures what is universal about the devices it describes

P6) A mobile phone model (a) allows specific RAM size options and (b) is a device model

P7) A mobile phone device (a) conforms to a mobile phone model, (b) has an IMEI, and (c) has a RAM size

P8) A mobile phone factory supports mobile phone models only

P9) A Huawei mobile factory (a) supports Huawei mobile phone models only, (b) keeps track of mobile phone devices it produces, and (c) constrains the IMEI of the mobile phone devices produced by the factory to start with '001'

P10) Factory124 (a) is a factory, (b) supports Huawei S400 and S500 mobile phone models, and (c) produced two S400 devices (S400_001, S400_002)

P11) S400 (a) is a mobile phone model and (b) has either 4GB or 8GB of RAM

P12) S400_001 (a) is a mobile phone device, (b) conforms to the S400 model, (c) has 4GBs of RAM, and (d) has '001468723648726' as its IMEI

P13) S400_002 (a) is a mobile phone device, (b) conforms to the S400 model, (c) has 8GBs of RAM, and (d) has '0018768768475638' as its IMEI

13.2 Model

The model consists of three levels with no optional styles or patterns in play.

13.2.1 Pan-Level Constraints and Types

Since only the default style is applied, there are no explicit pan-level constraints. The model also does not need any pan-level data type definitions.

13.2.2 Top Level (O_0): Types of Factories, Devices, and Companies

Figure 13.1 shows the top level that contains *FactoryAsModelSupporter*, *CompanyAsOwner*, and *DeviceModel*. The first two clabjects have a potency value

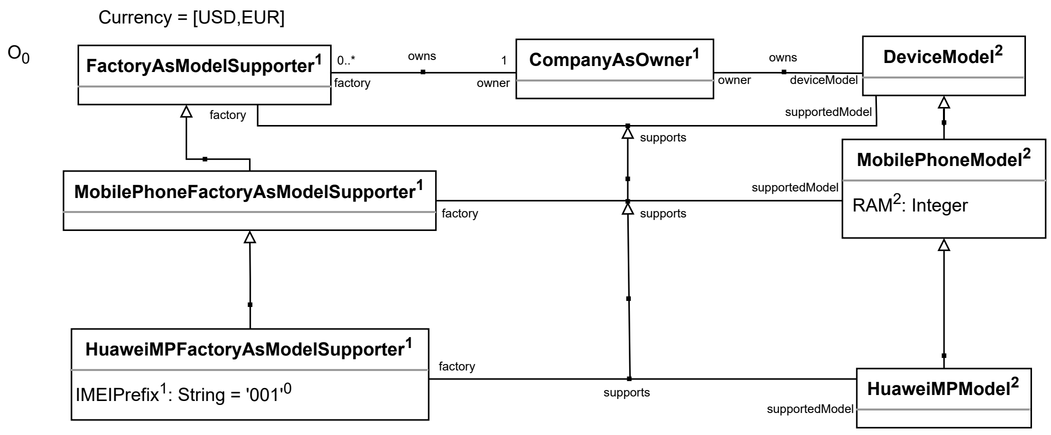


FIGURE 13.1: Level O_0 of the solution to the collaboration challenge

of 1, which specifies that instances of these clabjects are only present exactly one level below. The *DeviceModel* clabject has a potency value of 2 and therefore has direct instances in the middle level which in turn have instances at the bottom level. The top three clabjects in the middle level are *Factory*, *Company*, and *Device*, which are not ontologically typed (i.e. are not instances of a clabject at the level directly above). Clabjects *Factory* and *Device* are abstract (i.e. must not have any direct instances) which is represented by the fact that their potencies are 0. *Factory* is a superclass of *MP_Factory* and *HuaweiMP_Factory*, which is the only potent clabject in this inheritance hierarchy. *Device* is the superclass of *MP_Device*, which is an instance of *DeviceModel*. Every instance of *MP_Device* has to have an *IMEI* slot to identify the mobile hardware and a *RAM* slot. *MP_Device* is also an abstract clabject, as well as *HuaweiMP*. The abstract *produces* connection between the abstract clabjects *Factory* and *Device* is specialized via two inheritance relationships to reflect that only certain types of factories can produce certain types of devices. In this case, only *HuaweiMP_Factory* can produce devices of type *HuaweiMP_Device*.

13.2.3 Second Level (O_1): Concrete Device Models

Figure 13.2 shows the second level, O_1, of the solution to the collaboration challenge. Clabjects *S400* and *S500*, which represent device models, can be instantiated and are both connected to an instance of *FactoryAsModelSupporter* named *Factory124* and an instance of *CompanyAsOwner* named *Huawei*. *Factory124* and *Huawei* have the same linguistic names (i.e. the one appearing in the name compartment of clabject renderings) as respective individuals of

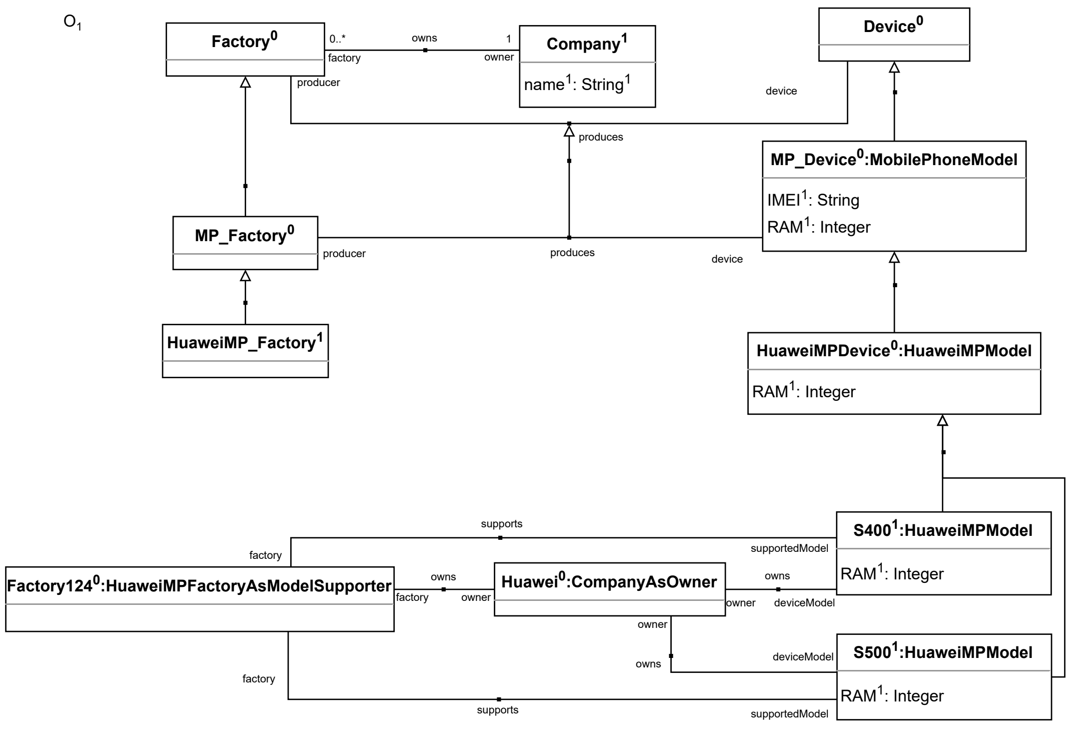


FIGURE 13.2: Level *O1* of the solution to the collaboration challenge

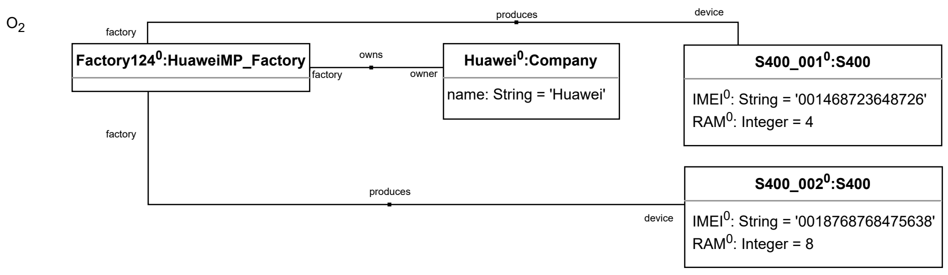


FIGURE 13.3: Level *O2* of the solution to the collaboration challenge

the clabjects *Factory* and *Company*. This models the fact that they represent the same real-world objects respectively, but at different levels of abstraction. In other words, the *CompanyAsOwner* and *Company* instances represent the same real-world object called "Huawei", while the instances of *FactoryAsModelSupporter* and *HuaweiMP_Factory* represent the same real-world object called "Factory124". This dual-level representation is designed to comprehensively cover the challenge specification while adhering to Melanee's strictness requirements that prohibit cross-level connections.

13.2.4 Third Level (O_2): Concrete Devices

Figure 13.3 shows the most concrete level of the solution to the collaboration challenge where the actual devices that are produced by *Factory124*, as an instance of *HuaweiMP_Factory*, are located.

13.3 Fulfillment of the Requirements

To ensure that the model is well-formed with respect to the dual-level representation of "Huawei" and "Factory124", two DOCL constraints are needed. These constraints, Constraint 13.1 and 13.2, not only ensure that a *CompanyAsOwner* instance is linked with a *FactoryAsModelSupporter* instance and a *Company* instance is linked with a *Factory* instance but also that they have to be connected to corresponding instances on their respective levels (via the *owns* connection). They are very similar in nature, the only difference is the context which points to a different starting point of the constraint to check the symmetry of the existence of the clabjects involved in the dual-level representation.

```
context Company(2_2)
inv: let companyName = self.#name# in Clabject -> select(clabject|clabject.
    isDeepOclTypeOf(CompanyAsOwner)) -> select(companyAsOwner|companyAsOwner.#
    getPotency()# = 0) -> select(companyAsOwner|companyAsOwner.#name# =
    companyName).factory -> collect(#name#) -> includesAll(self.factory ->
    collect(#name#))
```

CONSTRAINT 13.1: Matching name and associations from Company to CompanyAsOwner

```
context CompanyAsOwner(1_1)
inv: let companyAsOwnerName = self.#name# in Clabject -> select(clabject|
    clabject.isDeepOclTypeOf(Company)) -> select(company|company.#getPotency()#
     = 0) -> select(company|company.#name# = companyAsOwnerName).factory ->
    collect(#name#) -> includesAll(self.factory -> collect(#name#))
```

CONSTRAINT 13.2: Matching name and associations from CompanyAsOwner to Company

P1) *Company* (O_1) has (a) a *name* attribute and (b) connects to *Factory* via *owns*. (c) *CompanyAsOwner* (O_0) is connected to *DeviceModel* via *owns*

P2) *Huawei* is represented twice, once (a) as an instance of *Company* owning *Factory124*, and once (b) as an instance of *CompanyAsOwner* owning the *S400* and *S500* device models

P3) (a) *Factory* is connected to *Device* via *produces*, (b) *FactoryAsModelSupporter* is connected to *DeviceModel* via *supports*, (c) Constraint 13.3 ensures that a *Factory* only produces devices that are supported by the corresponding *FactoryAsModelSupporter*.

Constraint 13.3 ensures that every factory only produces the devices that conform to device models that it supports. For every device produced by a certain O_2-level factory, its type must be among the models that are supported by the O_1-level representation (*FactoryAsModelSupporter* instance) of that factory (with which it shares the same name).

```
context Factory(2_2)
inv: let factoryName:String = self.#name# in
        let factoryTypeRole = Clabject -> select(clabject|clabject.
    isDeepOclTypeOf(FactoryAsModelSupporter) -> select(clabject|clabject.#
    getPotency()# = 0) -> select(clabject|clabject.#name# = factoryName) in
        self.device -> forAll(device | factoryTypeRole.supportedModel ->
    includes(device.#getDirectTypes()# -> first()))
```

CONSTRAINT 13.3: Factory supported devices

P4) Devices are instances of device models

P5) Device models describe devices by being the types of the latter

P6) (a) Constraint 13.4 specifies the *RAM* options for a mobile phone model, (b) *MobilePhoneModel* inherits from *DeviceModel*. The constraint in Constraint 13.4 limits the values of *RAM* slots of *S400* instances to either '4' or '8'.

```
context S400(2_2)
inv: self.RAM = 4 or self.RAM = 8
```

CONSTRAINT 13.4: S400 RAM constraint

P7) (a) Mobile phone devices are instances of mobile phone models and the latter define (b) *IMEI* via inheriting from *MP_Device*, and (c) receive a *RAM* attribute from *MobilePhoneModel*

P8) Association *supports* between *MobilePhoneFactoryAsModelSupporter* and *MobilePhoneModel* is a specialization of the more general *supports* association, which restricts the supporting devices.

P9) (a) In level O_0 the *supports* association is specialized to restrict support between *HuaweiMPFactoryAsModelSupporter* and *HuaweiMPModel* instances (b) *HuaweiMP_Factory* inherits a *produces* association from *MP_Factory* (c)

Constraint 13.5 ensures that the IMEI starts with "001" This is guaranteed by the constraint in Constraint 13.5. From the context *HuaweiMP_Factory* this navigates to the produced devices and requires that the first three numbers of the IMEI have to be '001'.

```
context HuaweiMP_Factory(2_2)
inv: self.device -> forAll(IMEI.substring(1,3) = '001')
```

CONSTRAINT 13.5: IMEI constraint

P10) (a) *Factory124* at O_2 is an indirect instance of *Factory*, (b) at O_1, it *supports* the *S400* and *S500* mobile phone models, (c) and at O_2 it entertains *produces* links with *S400_001* and *S400_002*.

Constraint 13.6 ensures that every factory only produces the devices that conform to device models that it supports. For every device produced by a certain (bottom-level) factory, its type must be among the models that are supported by the middle-level representation (*FactoryAsModelSupporter* instance) of that factory (with which it shares the same name).

```
context Factory(2_2)
inv: let factoryName:String = self.#name# in
        let factoryTypeRole = Clabject -> select(clabject|clabject.
    isDeepOclTypeOf(FactoryAsModelSupporter) -> select(clabject|clabject.#
    getPotency()# = 0) -> select(clabject|clabject.#name# = factoryName) in
            self.device -> forAll(device | factoryTypeRole.supportedModel ->
    includes(device.#getDirectTypes()# -> first()))
```

CONSTRAINT 13.6: Factory supported devices constraint

P11) (a) *S400* is of type *HuaweiMPModel*, which specializes *MobilePhoneModel*, (b) the RAM choices are enforced via Constraint 13.4

P12) *S400_001*, (a) is an indirect instance of *MP_Device*, (b) is typed by *S400*, (c) has 4GB of RAM, and (d) the IMEI attribute has the specified value

P13) Analogously to **P 12)**, *S400_002* (a) is an indirect instance of *MP_Device* (b) is typed by *S400*, (c) has 8GB of RAM, and (d) the IMEI attribute has the specified value

13.4 Discussion

The solution to this challenge is not particularly interesting from the point of view of applying optional styles or patterns. No such optional styles or patterns are deployed meaning the modeling rules are governed by the strict modeling and characterization potency rules that come with the default style.

The most interesting aspect of the presented model is that it demonstrates how strict modeling may sometimes be too restrictive. To fulfill all the requirements the model had to deploy a "trick" to overcome the fact that the strict modeling style prohibits connections between clabjects at different levels, as essentially called for by the requirements. This trick is to represent each of the concepts *Factory* and *Company* by two clabjects, each representing a different perspective on the concept (i.e., from a different level). The use of this trick adds accidental complexity to the model, but also demonstrates the versatility of DOCL without which it could not be applied.

Chapter 14

The Warehouse Challenge

The most recent challenge from the MULTI community was the MULTI 2023 Warehouse Challenge, which focuses on the representation of product copies, product specifications, and product specification types with a particular emphasis on how to guarantee certain properties at the product level without fully determining them at higher levels. The solution has four levels and applies the universal ontological classification style (Constraint 9.1) as well as the Melanee default style (see Section 8.3). No patterns are used.

14.1 Requirements

This section summarizes the requirements of the Warehouse Challenge, defined in full in [80]. In contrast to previous challenges, the Warehouse Challenge is very specific about the particular properties that entities in the domain should possess and what values these properties should have. For example, product specification types must have *taxRate*, *currency*, and *introductionDate* properties.

Instances of product specification types are product specifications that specify the currency they are sold in. This currency should not be changed once the attribute has been set to a value, like “EUR”, “SEK” or “USD”. Instances of that product specification (i.e., that conform to it) are sold in that specified currency. Some products are not sold individually but rather sold in bulk, and have to be categorized as such. Although bulk product individuals should not be represented in the warehouse, information about the total number of individuals sold should be stored by bulk product specifications. Bulk products are sold as packages containing specified numbers of individuals.

For each product specification, the warehouse sets a standard sales price and a reduced sales price that must be lower than the standard sales price. The warehouse needs to keep track of the products sold, each product copy, and each product specification. The final price of a product is computed from the standard sales price, the reduced price (if one is present), and the tax rate. Each product should be able to report this final price.

Some product types are recommended by other product types. If a customer is in the process of buying a product of a specific type the warehouse system should be able to recommend a related product of another product type. For instance, mobile phones should recommend mobile phone cases but not other kinds of products.

R1) A Product Specification Type (a) has a tax rate, (b) specifies a currency, (c) has an introduction date, (d) describes copy specifications or bulk representations

R2) Book Spec (a) is a Product Specification Type, (b) has a tax rate of 7%, (c) specifies the currency EUR, (d) was introduced on 1 February 2003, (e) describes copy specifications

R3) Moby Dick (a) is a Book Spec, (b) has an SSP of EUR 9.95

R4) MB Copy 1 (a) is a copy of Moby Dick, (b) has an SSP of EUR 9.95

R5) MB Copy 2 (a) is a copy of Moby Dick, (b) has an SSP of EUR 9.95, (c) has been returned on 23 March 2023, (d) has a reduced price of EUR 1.95

R6) DVD Spec (a) is a Product Specification Type, (b) has a tax rate of 15%, (c) specifies the currency USD, (d) was introduced on 2 March 2004, (e) describes copy specifications

R7) 2001: A Space Odyssey (a) is a DVD Spec, (b) has an SSP of USD 19.95, (c) recommends haChi779

R8) DVD Player Spec (a) is a Product Specification Type, (b) has a tax rate of 15%, (c) specifies the currency USD, (d) was introduced on 3 April 2005, (e) describes copy specifications

R9) haChi779 (a) is a DVD Player Spec, (b) has an SSP of USD 99.99, (c) describes copies which have serial numbers

R10) Mobile Phone Spec (a) is a Product Specification Type, (b) has a tax rate of 15%, (c) specifies the currency SEK, (d) was introduced on 4 May 2006, (e) describes copy specifications

R11) Mate0815 (a) is a Mobile Phone Spec, (b) has an SSP of SEK 599.15, (c) recommends Matey

R12) MP Case Spec (a) is a Product Specification Type, (b) has a tax rate of 15%, (c) specifies the currency SEK, (d) was introduced on 5 June 2007, (e) describes copy specifications

R13) Matey (a) is a MP Case Spec, (b) has an SSP of SEK 17.95

R14) AA Battery Cell Spec (a) is a Product Specification Type, (b) has a tax rate of 15%, (c) specifies the currency NZD, (d) was introduced on 6 July 2008, (e) describes bulk product representations

R15) Energetic Plus (a) is an AA Battery Cell Spec, (b) has an SSP of SEK 1.50, (c) represents 271820 batteries which are sold as part of 10-packs

14.2 Model

This section presents our solution to the challenge. The deep model, which fulfills all the requirements, has four ontological levels, shown in Figures 14.1 through 14.4.

14.2.1 Level-Spanning Constraints

Like the solutions to all the other challenges, this challenge applies constraints to ensure that all clabjects in the levels derived from O_0 adhere to its rules. As mentioned above, the model applies one optional style – the universal ontological classification style defined in Chapter 9.2.1. The constraints that enforce these styles are therefore pan-level styles for this model. No pan-level datatypes are defined.

14.2.2 Top Level (O_0): Foundational Types

Figure 14.1 shows the top, most abstract, level which defines the foundational types in the domain. The most stable abstractions in the LML model

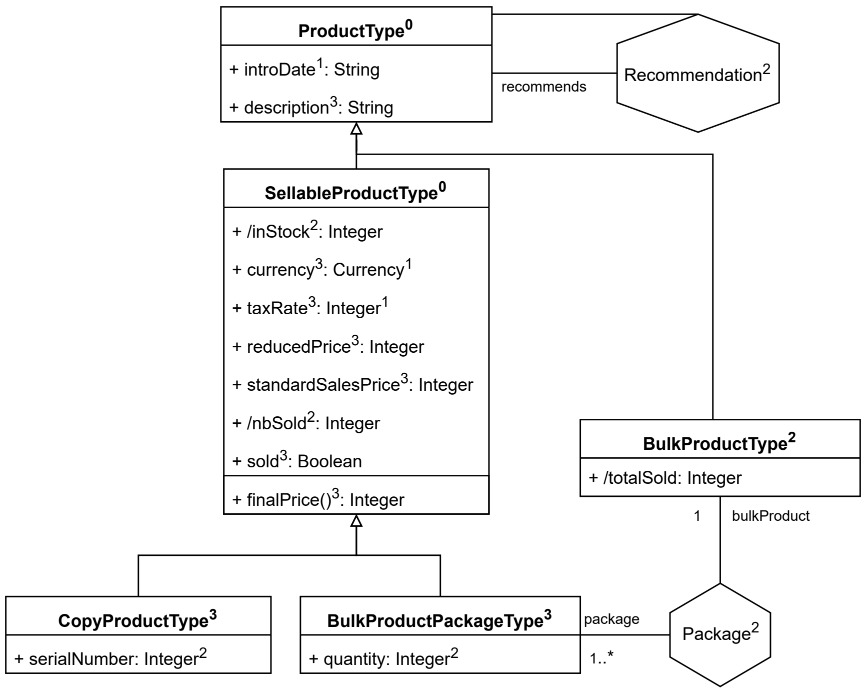


FIGURE 14.1: Foundational types at Level O_0

are the types that set up the context for other types and/or objects (i.e., clabjects) to represent a running incarnation of the warehouse that fulfills all the specified requirements. They therefore have two main jobs - (a) to characterize the model elements whose existence can change over time so that they can represent the appropriate domain properties and relationships, and (b) to ensure that only appropriate model elements can be added to the model that correctly represent a snapshot of the warehouse system at a particular point in time.

As shown in Figure 14.1, the solution has seven foundational types (i.e., clabjects), all contained in the top ontological level (O_0) – *ProductType*, which is an abstract class, *SellableProductType*, which is a subclass of *ProductType* and also abstract, *CopyProductType* and *BulkProductPackageType*, which are concrete subclasses of *SellableProductType*, *BulkProductType*, which is a subclass of *ProductType*, *Recommendation*, which is a connection clabject with *ProductType* as both its source and the target, and *Package* which is a connection clabject between *BulkProductType* and *BulkProductPackageType*. All seven foundational clabjects are highly stable components of the model that only need to be changed if the requirements of the warehouse system change.

ProductType is disjointly and completely specialized by *SellableProductType* and *BulkProductType* to reflect the fact that although bulk products are products they cannot be sold directly, they can only be sold indirectly via bulk

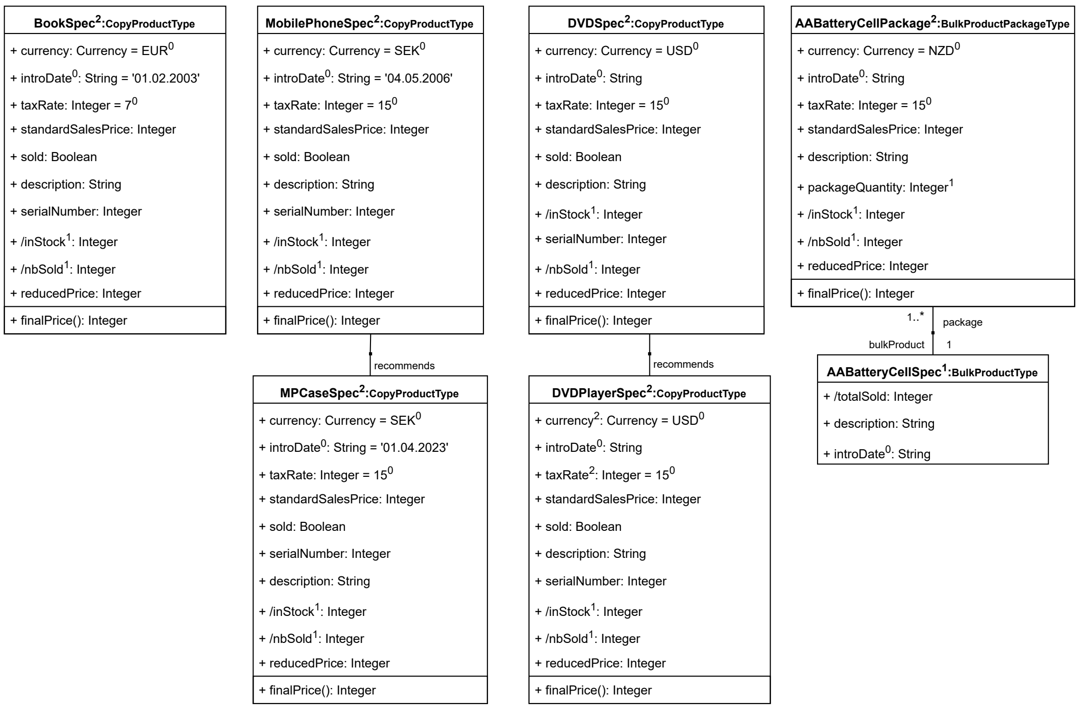


FIGURE 14.2: Product Specifications at the O_1 level

product packages. This is captured by creating *Package* connections between offspring of *BulkProductPackageType* and offspring of *BulkProductType*. Note, however, that because the potencies of *BulkProductType* and *Package* are lower than the potency of *BulkProductPackageType* (i.e., '2' rather than '3'), offspring of *BulkProductType* and *Package* cannot exist at the bottom level of the model, reflecting the fact that specific bulk product instances should not be stored in the warehouse.

Both connections in Figure 14.1 are shown in an exploded form. However, all of their instances at lower levels are shown in the imploded form as a dot. Melanee allows modelers to toggle between the two forms at any level as desired. Since the multiplicities of the *Recommendation* connection are the default (i.e., 1..*) they are not explicitly shown.

14.2.3 Second Level (O_1): Product Specifications

Figure 14.2 shows the second level of the deep model, where the next most stable abstractions in the model, product kinds, are defined. These represent major product categories like books, mobile phones, etc., that are not likely to change for many years. These reside at level O_1 of the model and, as shown in Figure 14.2, are instances of either *CopyProductType*, *BulkProductPackageType* or *BulkProductType*. All specified product kinds mentioned in the specific

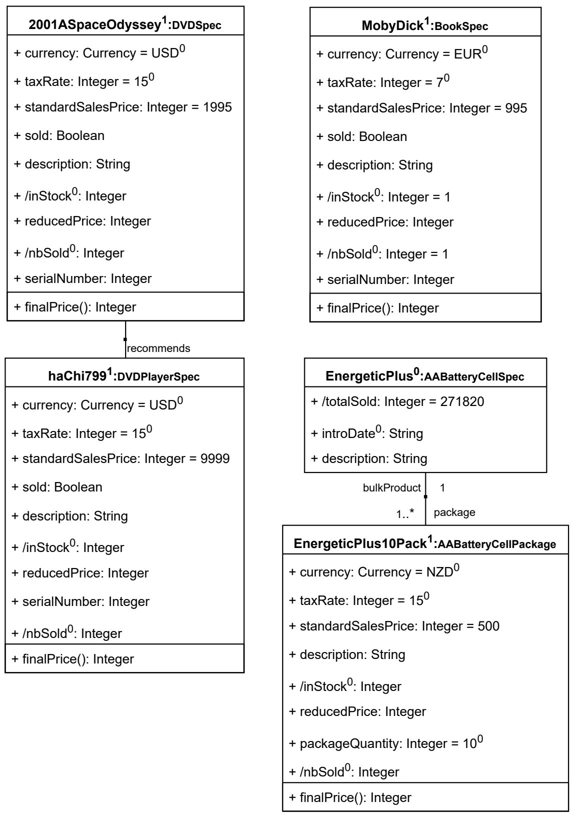


FIGURE 14.3: Products at the O_2 level

scenario outlined in the Warehouse Challenge description are included, with the prescribed attributes and associations.

In LML, connections are modeled as clabjects which can have all the usual features and relationships. In Figure 14.2, these connections are represented in collapsed form as a dot in the middle of the line between *MobilePhoneSpec* and *MPCaseSpec*, as well as between *DVDSpec* and *DVDPlayerSpec*. Connections can also be represented in expanded form using a hexagon symbol as shown in Figure 14.1.

Note that at this level, it is possible to choose what recommendation relationships can exist between sellable product specifications and what package relationships can exist between bulk products and bulk product package products. Once these connections have been selected at this level, the "Universal Ontological Classification" style ensures that all recommendation and package connections at the level below can only be between offspring of the appropriate kinds, as specified in the requirements.

14.2.4 Third Level (O_2): Products

Figure 14.3 shows the third level of the deep model, where the third most stable group of abstractions in the model, products, are defined. These are instances of the product specifications defined at the O_1 level. These are therefore defined at the O_2 level. The clabjects at this level represent specific products like specific books or specific mobile phone models. These change more frequently than the general product kinds of which they are instances.

The O_2 level contains all specified product kinds mentioned in the specific scenario outlined in the Warehouse Challenge description with the prescribed attributes and associations, namely – *MobyDick* (an instance of *BookSpec*), *2001ASpaceOdysey* (an instance of *DVDSpec*), *haChi799* (an instance of *DVDPlayerSpec*), *Mate0815* (an instance of *MobilePhoneSpec*), *Matey* (an instance of *MPCaseSpec*), *EnergeticPlus* (an instance of *AABatteryCellSpec*), and *EnergeticPlus10Pack* (an instance of *AABatteryCellPackage*). All of these clabjects have a common set of attributes as specified in *ProductType*. These instances have to possess these attributes due to LML's deep instantiation/-classification mechanism [19]. With the exception of *EnergeticPlus*, which is an offspring of *BulkProductType* and thus not directly offered for sale, all of the items are sellable. Note that the potency of *EnergeticPlus* is zero to ensure that it has no instances at the bottom level (O_2) as stated in the requirements.

14.2.5 Fourth Level (O_3): Product Instances

Finally, Figure 14.4 shows the fourth level of the deep model where the least stable group of objects in the model, specific product instances (i.e., individuals), are defined. These model elements are the actual items purchased by customers.

All specified product instances mentioned in the specific scenario outlined in the Warehouse Challenge description are included, with the prescribed attributes and associations, namely – *MBCopy1* and *MBCopy2* which are instances (i.e., copies) of the *MobyDick* book specification.

The attributes of *MBCopy2* indicate that it was returned on 23rd of April 2023 (i.e., is currently not sold) and has a *reducedPrice* of "1.95". Both books are sold in the "EUR" currency. The attributes of the other book, *MBCopy1*, indicate that it has been sold and is not currently in the warehouse (physically).

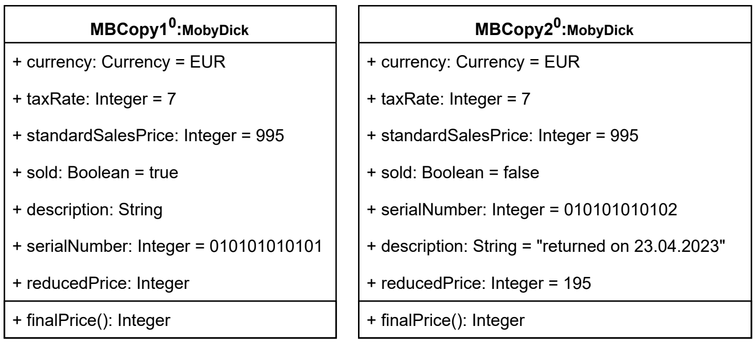

MBCopy1⁰:MobyDick
+ currency: Currency = EUR
+ taxRate: Integer = 7
+ standardSalesPrice: Integer = 995
+ sold: Boolean = true
+ description: String
+ serialNumber: Integer = 010101010101
+ reducedPrice: Integer
+ finalPrice(): Integer
MBCopy2⁰:MobyDick
+ currency: Currency = EUR
+ taxRate: Integer = 7
+ standardSalesPrice: Integer = 995
+ sold: Boolean = false
+ serialNumber: Integer = 010101010102
+ description: String = "returned on 23.04.2023"
+ reducedPrice: Integer = 195
+ finalPrice(): Integer

FIGURE 14.4: Product copy instances at the O_3 level

14.3 Fulfillment of the Requirements

R1) Figure 14.1 shows the level that contains the clabject *ProductType* that has an (c) *introductionDate* attribute and a subclass *SellableProductType* that (a) contains the attributes *taxRate* and (b) *currency*. The clabject *CopyProductType*, *BulkProductType*, and *BulkProductPackageType* (d) describe copy and bulk specification respectively.

R2) The *BookSpec* clabject on level 1 (Figure 14.2) (e) is of type *CopyProductType* and an indirect (a) instance of *ProductType*. The *taxRate* attribute (b) is set to 7% and to able to change in the lower levels due to the mutability value '0'. The (c) currency is EUR and the (d) *introductionDate* attribute is set to "01.02.2003".

R3) The *MobyDick* clabject at level 2 is (a) an instance of *BookSpec* and (b) the *standardSalesPrice* attribute has a value of '995', which is the price in cents (or the smallest unit in a currency).

R4) The *MBCopy1* clabject at level 3 (a) is an instance of *MobyDick* and (b) the *standardSalesPrice* is '995'.

R5) The *MBCopy2* clabject at level 3 (a) is an instance of *MobyDick* and (b) the *standardSalesPrice* is '995'. The *description* attribute (c) states that the book was returned on 23 March 2023 and the (d) *reducedPrice* attribute is set to '195'.

R6 The *DVDSpec* clabject at level 1 (e) is a direct instance of *CopyProductType* and (a) an indirect instance of *ProductType*. The *taxRate* (b) attribute is set to

15% and the *currency* (c) attribute is of type USD. The *introductionDate* (d) attribute is set to “02.03.2004”.

R7) The *2001ASpaceOdyssey* clabject at level 2 (a) is an instance of *DVDSpec* and the *standardSalePrice* (b) attribute is set to ‘1995’. This clabject is (c) connected to the *haChi779* clabject which it recommends via the *Recommendation* connection.

R8) The *DVDPlayerSpec* is (e) a direct instance of *CopyProdcutType* and (a) an indirect instance of *ProductType*. The *taxRate* (b) attribute is set to 15% and the *currency* (c) is set to USD. The *introductionDate* (d) is set to “03.04.2005”.

R9) The *haChi779* clabject at level 2 is (a) an instance of *DVDPlayerSpec* and the *standardSalesPrice* (b) attribute is set to ‘9999’. The *serialNumber* attribute (c) is present in the clabject.

R10) The *MobilePhoneSpec* clabject at level 1 (e) is a direct instance of *CopyProductType* and (a) an indirect instance of *ProductType*. The *taxRate* (b) attribute is set to 15% and (c) the *currency* attribute is of type SEK. The *introductionDate* (d) attribute is set to “04.05.2006”.

R11) The clabject *Mate0185* at level 2 is (a) an instance of *MobilePhonSpec* and (b) the *standardSalesPrice* is set to ‘59915’. This clabject is (c) connected to the *Matey* clabject via the *Recommends* relationship.

R12) The clabject *MPCaseSpec* at level 1 (e) is a direct instance of *CopyProductType* and (a) an indirect instance of *ProductType*. The *taxRate* (b) attribute is set to 15% and (c) the *currency* is of type SEK. The *introductionDate* (d) is set to “05.06.2007”.

R13) The *Matey* clabject is (a) an instance *MPCaseSpec* and (b) the *standardSalesPrice* is set to ‘1795’ and the currency is of type SEK.

R14) The clabject *AABatteryCellSpec* at level 1 is (e) a direct instance of *BulkProductType* and (a) an indirect instance of *ProductType*. The *taxRate* (b) attribute is set to 15% and the (c) *currency* attribute is of type NZD. The *introductionDate* (d) attribute is set to “06.07.2008”.

R15) The *EnergeticPlus* clabject at level 2 is (a) an instance of *AABatteryCell-Spec*. The *standardSalesPrice* attribute is not present in this clabject but rather in the *EnergeticPlus10Pack* which is an instance of *AABatteryCellPackage*, which, in turn, is an instance of *BulkProductPackageType* and thus an indirect descendant of *SellableProduct* that has the *standardSalesPrice* attribute. In the package representation, the *standardSalesPrice* is (b) set to '150' and the *currency* attribute is set to SEK. The derived attribute *totalSold* (c) computes the total number of sold batteries (through packages).

14.3.1 Attributes

The attributes we have not yet discussed in the previous list of the fulfillment of the requirements are discussed in this section.

The *SellableProductType* clabject specifies attributes and methods that are specific to all sellable products. Specifically, these are:

- *inStock* of type Integer. This attribute is a derived attribute that stores the number of product instances of a given product specification (e.g., *Book*) or a given product (e.g., *MobyDick*) that are currently in stock, at the O_1 and O_2 levels respectively. Its durability and mutability values are '2'. The DOCL expression defining how the values of this attribute are calculated can be seen in Constraint 14.1.

- *sold* of type Boolean. This attribute has a value of 'true' if the individual product is sold and no longer physically in the warehouse. The durability and mutability values equal the potency value.

- *nbSold* of type Integer. This attribute is a derived attribute that stores the number of sold product instances of a given product specification or a given product, at the O_1 and O_2 levels respectively. Its durability and mutability values are '2'. The DOCL expression defining how the values of this attribute are calculated can be seen in Constraint 14.2.

- *finalPrice()* is an Integer valued method that computes the price the customer has to pay for a product from the standard sales price, reduced sales price, and the tax rate. Its durability and mutability values are '3'. The DOCL expression defining how the output of this method is calculated can be seen in Constraint 14.4.

Only three other attributes are defined at the (O_0) level – *serialNumber*, *quantity* and *totalSold* which belong to *CopyProductType*, *BulkProductPackageType* and *BulkProductType* respectively.

- *serialNumber* of type Integer. With a durability of '3' and mutability of '2', this attribute stores the serial number of *CopyProductType* offspring.

- *quantity* of type Integer. With a durability of '3' and mutability of '2', this attribute stores the number of items of the connected *BulkProductType* that an offspring of *BulkProductPackageType* contains.

- *totalSold* of type Integer. With a durability of '2' and mutability of '2', this attribute indicates the number of items of a given bulk product type (e.g., *AABatteryCellSpec*) and bulk product (e.g., *EnergeticPlus*) have been sold, at the O_1 at the O_2 levels respectively. Since *BulkProductType* items cannot be sold individually, but only through *BulkProductPackageType* offspring, this is a derived attribute that is calculated from the number of corresponding *BulkProductPackageType* offspring sold. The DOCL expression for deriving this value is shown in Constraint 14.3.

The complete specialization of *ProductType* by *CopyProductType*, *BulkProductPackageType* and *BulkProductType* reflects the fundamental categorization of all products as being either bulk products, bulk product packages or copy products.

Derive Constraints

The O_0 level defines three derived attributes – the *inStock* and *nbSold* attributes of *SellableProductType* and the *totalSold* attribute of *BulkProductType*. To ensure that all offspring of *SellableProductType* and *BulkProductType* at the O_1 and O_2 levels have the appropriate values for these attributes three deep derive constraints are needed.

The first derive constraint, Constraint 14.1, for the *inStock* attribute has *SellableProductType* as its context. For any offspring, s, of *SellableProductType* for which the attribute exists (i.e., offspring at the O_1 and O_2 levels, but not the O_3 level), the constraint starts by collecting all clabjects of potency 0 that are also offspring of s, and then selects the subset whose *sold* attribute has the value false.

```
context SellableProductType::inStock:Integer (0,2)
derive: Clabject -> select(clabject| clabject.#getPotency()# = 0 and clabject.
    isDeepInstanceOf(self)) -> select(instance|instance.sold = false) -> size()
```

CONSTRAINT 14.1: value of the InStock derived attribute of sellable products

The second derive constraint, Constraint 14.2, for the *nbSold* attribute also has *SellableProductType* as its context. It is essentially the inverse of Constraint 14.1 in that it calculates the number of sold product instances of offspring of *SellableProductType* at the O_1 and O_2 levels, rather than the number in stock. The constraint has essentially the same structure as Constraint 14.1, except that it selects the subset whose *sold* attribute has the value true rather than false.

```
context SellableProductType::nbSold:Integer (0,2)
derive: Clabject -> select(clabject|clabject.isDeepInstanceOf(self) and
    clabject.#getPotency()# = 0) -> select(instance|instance.sold = true) ->
    size()
```

CONSTRAINT 14.2: Value of the nbSold derived attribute of sellable products

The third derive constraint, Constraint 14.3, for the *totalSold* attribute has *BulkProductType* as its context. Since *BulkProductType* can only have offspring at the O_1 and O_2 levels, this constraint calculates appropriate values of the *totalSold* attribute for all offspring of the *BulkProductType*. For any offspring, b, of *BulkProductType* the constraint starts by collecting all clabjects of potency zero that are also offspring of b and then uses an "iterate" operation to accumulate the number of items sold via all *BulkProductPackageType* offspring at level O_2 connected to b by an offspring of the *Package* connection. Essentially, the constraint sums up all the items sold via the related bulk product packages.

```
context BulkProductType::totalSold:Integer (0,2)
derive:  Clabject -> select(clabject|clabject.isDeepInstanceOf(self) and
    clabject.#getPotency()# = 0) -> iterate(p; acc = 0 | acc = acc + (p.package
    .quantity * p.package.nbSold) )
```

CONSTRAINT 14.3: Value of the totalSold derived attribute of bulk products

Body Constraints

The method *finalPrice* calculates the price of a sellable product depending on the value of the *reducedPrice* attribute, or the lack thereof. The following body constraint, Constraint 14.4, specifies how this value is calculated. If the value of the *reducedPrice* is undefined, the *standardSalesPrice* is the basis of the tax calculation that gives rise to the final price that customers have to pay.

```
context SellableProductType::finalPrice():Integer (3,3)
body: let value:Integer = 0 in
    if self.reducedPrice.oclIsUndefined()
    then value = self.standardSalesPrice +
        self.standardSalesPrice * (taxRate / 100)
    else value = self.reducedPrice + self.reducedPrice * (taxRate / 100)
    endif
```

CONSTRAINT 14.4: Body constraint specification for the final price

14.4 Discussion

Although the solution to this challenge does not apply many optional styles or patterns, it nevertheless possesses some interesting aspects. The first is that, in contrast to the previous challenge, Melanee's default styles appear to provide an optimal amount of rigor for the needs of the domain. In particular, the strict modeling and characterization potency combination allows all the requirements to be concisely modeled with no accidental complexity (such as double representations of concept), whilst still ensuring that unwanted constructs are prohibited. The default style was complemented with the optional universal ontological classification style to prevent modelers from introducing ontologically untyped clabjects at lower levels.

The second interesting aspect of the model is the use of three derive constraints and one body constraint to deal with unusually demanding requirements. The derive constraints were used to ensure that all offspring of *SellableProductType* and *BulkProductType* at the O_1 and O_2 levels have the appropriate values, while the body constraint was used to model the functionality of the *finalPrice* method which calculates the price of a *sellable* product depending on the value of the *reducedPrice* attribute, if present. All these different use cases demonstrate the versatility of the DOCL constraint language.

Part VI

Significance

The final part contains two chapters that place the technology developed in this thesis in the context of related work and summarise its relevance. The first chapter describes various kinds of related work, including other variants of, and tools supporting, the OCL, alternative constraint languages, and the main multi-level modeling languages beyond the Melanee tool used in the thesis. The second chapter summarizes the achievements of the thesis in terms of how well it fulfills the defined requirements and solves the identified challenges, examines the degree to which the presented work validates the hypothesis, and identifies significant potential lines of future work.

Chapter 15

Related Work

This chapter summarizes academic work related to this thesis. The first section deals with other constraint languages and their respective implementations. The second section presents other multi-level tools and how they handle constraints in their languages. The third section gives a brief overview of other approaches to view-based modeling.

15.1 OCL Variants and Tools

There are different implementations of the OCL standard and different tools supporting them. This section presents an overview of the tools that are discussed and used in the literature and also the most important open-source implementations of the OCL.

15.1.1 Eclipse OCL ™

The Eclipse OCL™ [121] implements standard OCL for EMF-based models. This version of OCL works on Ecore models and UML models in Eclipse. Queries and OCL expressions on those models are made accessible through an API while being defined in Ecore as an OCL abstract syntax model. Expressions are parsed with the help of an LARL parser generator that accepts (E)BNF (extended Backus-Naur Form) rules.

15.1.2 DresdenOCL

DresdenOCL is an open-source framework for OCL development and analysis. It provides a range of tools and libraries for working with OCL, including an OCL parser and interpreter, a model-driven testing framework, and a constraint analyzer [45, 46]. The goal of DresdenOCL is to provide a comprehensive and user-friendly platform for OCL development and analysis, with a focus on model-driven engineering and software testing. It is designed to be extensible and customizable, allowing users to add their own plugins and tools as needed.

One of the key features of DresdenOCL is its support for OCL testing and validation. The framework includes a model-driven testing framework that enables users to generate test cases automatically from OCL constraints and test their models against these constraints [73, 104]. It also includes a constraint analyzer that can be used to analyze OCL constraints and detect errors and inconsistencies. The interpretative approach is used when the model has to be simulated (or animated). This can only be done when the model is "stateful" (in terms of activity diagrams or state charts) or the constraints describe the behavior of the execution semantics.

DresdenOCL is written in Java and is compatible with a range of modeling tools and frameworks, including Eclipse, EMF, and UML.

15.1.3 XOCL im FMML^x

This constraint language represents OCL expressions in an XML-based format. The meta-model for XOCL is based on a UML class diagram with OCL constraints that define rules on how the constraints can interact with the rest of the class diagram [102]. The XML schema for XOCL is defined in XMLS, which allows the designer to specify the structural composition of the XML document.

XOCL is a language that is defined "in itself" (i.e., is self-descriptive and self-contained [35]). It is also a language that can be characterized as being meta-circular [39]. The mechanism that allows for a language to be defined in itself is called "bootstrapping" [100] and offers extension points to extend or alter the syntax and semantics of the language. XOCL also offers reflective features [59] that allow for access to the meta-level.

In multi-level modeling, the FMML^x tool uses this language to express constraints [50] based on XModeler^{ML} [38].

15.1.4 OCL$_R$

Chapter 5 relies heavily on the work of Draheim [48] which extends OCL with reification (which we refer to as introspection or reflection in the narrow sense) and reflection (which we refer to as intercession or reflection in the wider sense). Constraints defined in Draheim's OCL dialect, called OCL$_R$, are defined at the M1 level and can navigate up and down the classification hierarchy of the UML 4-layer infrastructure (see Figure 2.2). The abstract syntax of OCL_R is defined at level M2.

The goal of OCL_R is to make models robust against changes/updates at the M1 level (i.e., to provide quality assurance for system design). The supporting techniques are called "Subtype Externalization" and "Powertype Externalization". The former relates to the style definitions we defined in Chapter 10.2.1, where the inheritance patterns are defined, whereas the latter relates to Chapter 10.3 where the categorization patterns that lead to the application of the powertype pattern are defined.

In OCL_R, the context of an expression is essentially irrelevant because the set of desired model elements a constraint applies to can be established by querying for a certain type or a certain property of a type. The language uses a practical reification approach and leverages the four-level infrastructure to achieve well-formedness. In contrast, DOCL is implemented by reusing the OCL syntax and most of its semantics. Because DOCL is defined in its own grammar and interpreter, therefore, there are no restrictions on the kinds of changes and additions that can be made.

15.1.5 $\text{OCL}^\#$

More recently, Steimann, Clarisó, and Gogolla [113] have proposed a new interpretation of collection types in OCL. They present a core calculus that solves four key issues they have identified in the OCL specification.

The first issue is the *impedence mismatch* that stems from the difference in navigation semantics in UML and OCL. The UML uses multiplicities and OCL collections which makes the navigation, depending on the multiplicity values of the connections, not null-safe. The semantics dealing with collection operations on null values and *OclInvalid()* types are not well-defined. That leads to the second issue that *null* itself is an object in OCL that should denote the absence of any value, but instead, it can be an element of collections.

The third issue deals with the self-referentiality of OCL, which means that OCL is defined in terms of itself. The fourth and last issue describes the typing problems with the collections library in the current OCL specification. They argue that *Collection* as the common supertype for *Set* and *Bag* has very little semantic meaning and aim to redefine the OCL type system in that regard so that collection types are accurately defined. They realized a new OCL specification, called $\text{OCL}^\#$, by introducing what they call *ctype* which is a newly implemented type system for collections, that supports not only *Set*, *OrderedSet*, *Bag*, and *Sequence* but also nested collections. They made the navigation type safe in the sense that "everything is a collection", removed the self-referentiality from the specification, and reified the null object to "no object".

15.1.6 OCL$_{\text{UNIV}}$

This version of OCL [97, 98] is motivated by the fact that the UML/OCL schemas are undecidable [23] in terms of reasoning on them. Therefore, the authors propose to build a subset of OCL that conserves the decidability of first-order logic expressions. By avoiding the *exists* operation and negation OCL$_{\text{UNIV}}$ was designed to be checkable by means of SQL queries in finite time. OCL$_{\text{UNIV}}$ also preserves decidability even when reasoning in *weakly acyclic* UML class diagrams. DOCL has the same decidability problems as regular OCL.

15.2 Alternative Constraint Languages

Given the central role of the UML in visual modeling, its associated constraint language, OCL, is by far the most widely used language for increasing the precision of models. However, various other constraint languages are also important. This section summarises these different languages.

15.2.1 The Epsilon Language Family

One particular language in the Epsilon language family, the Epsilon Object Language (EOL) [76], is used for similar purposes as OCL. In particular, using EOL it is possible to access and modify multiple models and verify their integrity. Individual model elements, as well as complete models, can be created, updated, and deleted. All languages in the Epsilon family use a common core for their expressions, which makes switching between languages in the family (e.g., to define transformations) very easy.

Although the Epsilon languages have strong support for the EMF built in, they can be used with other modeling platforms. They are meta-model independent and offer interfaces to tool builders to facilitate seamless integration [76]. Also, unlike OCL, the languages of the Epsilon family can also be used stand-alone, i.e., without being tied to another modeling language

[115]. The MetaDepth tool [41] uses the EOL, the ETL (Epsilon Transformation Language) for in-place transformations, and the EGL (Epsilon Generation Language) to express constraints, transformations, and generation rules. At the time of writing, there are no papers published about the reflective capabilities of MetaDepth.

15.2.2 Datalog

Datalog [57] is a declarative logic programming language used for querying and manipulating relational databases. It is based on the concept of first-order logic and allows users to express queries and rules using a syntax similar to that of the Prolog [64] programming language.

Datalog programs consist of a set of rules that define relationships between data items, and queries that retrieve information from the database based on these rules. The language is particularly useful in applications that involve complex data processing and analysis, such as data mining, machine learning, and knowledge representation. One of the key features of *Datalog* is that it supports recursive queries, which enable users to define rules that refer to themselves. This allows for the efficient processing of complex data structures and the identification of patterns and relationships that might not be apparent using other programming languages or database systems.

Datalog and OCL are both declarative languages used for expressing constraints and rules on data. However, there are some important differences between the two languages. For example, *Datalog* is primarily used for querying and manipulating relational databases, whereas OCL is a modeling language used in software engineering to specify constraints and conditions on object-oriented models. Also, *Datalog* is based on the concept of first-order logic and uses a syntax similar to Prolog, whereas OCL is based on the UML and has a syntax that is similar to a programming language. In terms of functionality, *Datalog* is designed to support recursive queries, while OCL supports operations on collections and set-valued attributes.

Despite these differences, both *Datalog* and OCL are useful for expressing complex constraints and rules on data. They can be used together in certain contexts, such as in data-intensive software systems where both modeling and querying of large datasets are required.

15.3 Multi-level Modeling Approaches

This section presents alternative multi-level modeling to the Melanee/LML approach described in this thesis. Some of the approaches use the OCA while others do not. Similarly, some are level-adjuvant while others are level-blind.

15.3.1 MetaDepth

MetaDepth, developed by de Lara and Guerra [41], is a deep-modeling tool that supports textual modeling over an arbitrary number of ontological levels and the dual instantiation of *Clabjects* [41]. The tool utilizes EOL [76], which is partly built on OCL, to define constraints. As explained in the previous section, EOL is part constraint language and part action language.

MetaDepth's version of EOL has two main changes to support multi-level modeling. The first is to be able to assign a potency to a constraint which shows the level at which the constraint should be evaluated. If the potency is 1 then the constraint is evaluated one level below the level at which the constraint is defined. The second enhancement is that the constraints can access methods and attributes of the linguistic dimension. Properties of the linguistic dimensions are accessed like properties from the ontological dimension. If there is a name collision between attributes or methods from the linguistic and ontological dimensions and the user wants to access the property from the linguistic dimension the prefix "^" can be used to indicate the dimension switch [42]. DOCL has a similar dimension switch to the linguistic dimension, but the user has to indicate that switch explicitly in contrast to the MetaDepth approach, where constraints access the properties of the linguistic dimension by default.

The idea of defining potencies on constraints to indicate the level the constraint should be evaluated on is to some extent consistent with the idea of potency in general in the deep-modeling context. However, it would be more useful to indicate a range of levels over which the constraint should be valid because the potency defined on a *Clabject* indicates how many levels below it can have instances.

15.3.2 MultEcore

MultEcore is a tool based on EMF that aims to combine the best of the fixed-level and multi-level metamodeling approaches [89]. Like Melanee, MultEcore utilizes the Eclipse ecosystem built around the EMF, but instead of using GMF as the editor, MultEcore employs *Sirius*. The MultEcore approach to the linguistic metamodel is "loose". Every class in a MultEcore model has to have an ontological type but not necessarily a linguistic type.

In recent publications, the MultEcore tool support has been extended with a branching mechanism that can be used when multiple instance levels share the same meta-levels and a behavior description, as well as a constraint language for static and dynamic semantics [105]. MultEcore also provides tooling for the composition of domain-specific languages inside the multi-level modeling framework and applies it to the example of executable colored Petri-Nets [106].

15.3.3 DLMA

Dynamic Multi-Layer Algebra (DLMA) [117, 118] aims to support a top-down refinement approach to MLM. DLMA consists of the core, where the formal definitions and model management functions reside, and the bootstrap, which contains a set of reusable entities. In theory, the bootstrap can be changed and altered to a modeler's needs in order to satisfy the domain specifications. The core is interpreted by an Abstract State-Machine (ASM). The DLMA approach does not use levels to group entities into an (onto-)logical union – in other words, it is essentially "level-blind". However, the notion of levels could be integrated as a concept into the bootstrap so that the approach becomes "level-adjuvant" [16]. In terms of constraints, DLMA has a built-in language called *DLMAScript* to validate models, which is not based on OCL. Users can create custom validation formulae to make precise statements about the domain [21].

15.3.4 FMML^x

The FMML^x tool has its foundation in the XModeler^{ML} execution environment, which in turn, is based on the reflective "golden braid" metamodel of *XCore* [36, 37]. In XCore, every class that is instantiated is also at the same time a subclass of the meta-class *Class*. Since FMML^x is a monotonic extension of XCore, the meta-class *Class* inherits also from *Object*. In this way, everything that is created in a model is essentially an object. Properties of classes can be defined as *intrinsic*, which means they have to receive a specific value somewhere in the instantiation chain. The modeler can also specify where, and on which level, the attribute value has to be specified [50].

Connections, called associations in FMML^x, can connect classes from different levels but also classes from the same level. FMML^x does not use potency to control the instance-of relationships but uses the "order" concept which indicates at which level the class is located. Because they are founded on the XModeler^{ML}, every model of FMML^x is executable and can be manipulated by the user through XModeler^{ML}'s GUI.

15.3.5 Nivel

Nivel is another deep modeling framework that was created by Asikainen and Männistö [9]. The framework enables the user to define *cardinality constraints* that affect instances of associations holding values for cardinality and potency. Nivel models have formal semantics by translation to the WCRL, a general-purpose knowledge representation language.

Asikainen and Männistö state that "Nivel defines no constraint language of its own" [9] and the cardinality constraint construct is the only possibility to define constraints on the model or elements of the model. According to the authors, adopting WCRL as the constraint language (without the translation of the model) for Nivel would cause a number of problems and is not desired. They claim that any user who is familiar with Nivel is assumed to be familiar with the WCRL and able to write constraints in it.

15.3.6 Nivel 2

The multi-level modeling web environment called *nivel2* [10], is built on top of a relational database management system and supports the creation, manipulation, and visualization of models at different levels of abstraction. The authors argue that existing multi-level modeling tools often suffer from limitations such as a lack of support for multiple views, difficulties in managing large models, and limited collaboration and version control capabilities. Nivel2 addresses these limitations by providing a web-based interface that allows for the creation and management of multi-level models in a collaborative and distributed environment. The authors describe the architecture and implementation of *nivel2* and present a case study of its application to a large-scale modeling project in the field of energy management. Overall, *nivel2* provides a tool for MLM that can support complex modeling scenarios and enable collaboration and version control among modelers and stakeholders.

15.3.7 DLM

Kühne [78] introduced the idea of supporting multiple ontological dimensions in a multi-level model, rather than just the usual two. These dimensions are intended to be orthogonal to each other and can overlap so that clabjects can be part of multiple dimensions. The downside is that the language has to allow clabjects to have multiple direct types as well as multiple potency values depending on the context of the dimension they are in. The rules of potency have to apply in each dimension. The main advantage of DLM is that it can allow cross level connections without losing all the built-in error avoidance provided by strictness. Since DLM is otherwise based on the same core ideas as LML, DOCL can easily be used to add precision to DLM models. However, full support for multiple dimensions would require DOCL to be equipped with further features.

15.3.8 DeepTelos

Jeusfeld and Neumayr [70] developed DeepTelos as an extension to Telos [93]. Telos in turn is implemented in Conceptbase [69] by means of 30 axioms defined on top of the base predicate [68]. These axioms govern the rules of instantiation, attributes, associations, and specializations. Deep Telos is defined by 6 more deductive rules and constraints that are specified on top of the 30 axioms that are needed to define Telos [68].

Deep Telos is a so-called level-blind approach to multi-level modeling because it does not have a unique concept of “level” to organize the clabjects or other model elements. Since levels “emerge” in their models as part of the classification hierarchy, Deep Telos can be characterized as a “loose” approach to MLM.

Deep Telos has built-in support for constraints (as in DLMA). This allows users to define constraints in predicate logic [68].

15.3.9 MLT

Many approaches to multi-level modeling have been proposed to date. However, some approaches are ad-hoc and lack a formal grounding, which limits their effectiveness in capturing the complexity of real-world systems. MLT (Multi-level theory) [29] stands for a well-founded theory that is based on the principles of ontology, formal semantics, and logic, and aims to provide a formal and rigorous framework for multi-level modeling. MLT defines a set of concepts, relations, and axioms that allow for the representation, integration, and reasoning of models at different levels of abstraction.

15.3.10 UFO-MLT

UFO-MLT is a combination of the Unified Foundational Ontology and the aforementioned MLT multi-level modeling approach [31]. MLT is a conceptual modeling theory that is based on the notion of “classes” and “types” and also includes the notions of “type” and “ individual” [30] from the ontology domain. In comparison to UFO or OntoUML, MLT can support the MLM principle that “types” also can have “types” and are instances of those more abstract types.

Chapter 16

Conclusion

The range of MLM languages and tools summarized in the previous chapter demonstrates the high level of activity and interest in the MLM research community but also reveals symptoms of two significant weaknesses in the current state-of-the-art –

1. The large number and high heterogeneity of the core MLM languages and tools currently available, and the diverse range of modeling approaches and concepts they offer. This heterogeneity spreads the resources of the MLM community thinly, diluting the quality of individual tools and making it difficult for users to select which approach to use. Moreover, users who have learned the syntax and semantics of a particular language and its associated tooling have a high learning curve to use an alternative MLM approach that may better match the needs of a particular application.

2. The much smaller and limited range of constraint languages that have been developed to partner the core, graphical MLM languages. Moreover, the partner languages that do exist usually only have limited support for MLM concepts (i.e., minimal multi-level awareness) and/or are highly technical languages that are difficult for mainstream developers to learn and apply.

Together these weaknesses reduce the usability of MLM technology and the precision of multi-level models. The technology developed by this thesis helps overcome both problems by (a) enhancing OCL, the most user-friendly and widely-used constraint language for the UML, to be a fully multi-level adjuvant partner of the LML, and by (b) including reflection capabilities that allow many of the variations and idiosyncrasies of different MLM languages to be captured as styles and/or patterns of usage of a common underlying core MLM approach.

16.1 Problems and Requirements

The significance of the aforementioned weaknesses, and the nature of the required advances needed to address them, were outlined in Chapter 1. This section summarizes how the DOCL technology presented in this thesis fulfills these requirements.

16.1.1 Requirements

R1 and R2: The first two requirements basically describe the core functionality and usability properties the language is required to have. The first requirement calls for a constraint language, founded on set theory and first-order logic, that facilitates the definition and checking of precise constraints on the concepts expressed in a model, while the second requirement calls for a language that does this in a way that is easy for mainstream software engineers and domain experts to use. DOCL fulfills these requirements as it is an extension of OCL which was expressly designed to have these features.

R3: The third requirement calls for a language in which multi-level constraints are written as a conservative extension of two-level constraints. This means that traditional two-level constraints are a special case of multi-level constraints. DOCL has been carefully designed to fulfill this requirement by defining the absence of an explicit multi-level aware expression in a constraint, such as level-range specifications or meta-level introspection expressions, to have the classic, two-level meaning. Thus, for example, when no level-range specification is present in a constraint, the range of the affected levels defaults to the level immediately below, as in classic OCL.

R4: The fourth requirement calls for a language that is not just multi-level aware, but also multi-level adjuvant. This means it should actively support the definition of deep constraints in a level-agnostic way. DOCL fulfills this requirement by using level-agnostic concrete syntax and providing a large selection of new operators that provide ontological introspection information in a uniform and seamless way, regardless of where (i.e., from what level) the introspection is taking place.

R5: The fifth requirement calls for a language that supports reflection in a way that is not only compatible with the OCA but also supports seamless introspection in both dimensions. DOCL fulfills this requirement by explicitly supporting two orthogonal introspection capabilities - ontological introspection, which allows a constraint to access information from higher ontological levels, and linguistic introspection which allows a constraint to access information from the linguistic metamodel (i.e., the PLM). The concrete syntax used to express these dimension "switches" is level-agnostic and can be nested in arbitrary ways.

16.1.2 Problems

By fulfilling all the aforementioned requirements, DOCL provides a concise and user-friendly solution to the three basic problems outlined in Chapter 1.

P1: The first problem of rigid modeling styles is a symptom of the fact that the majority of MLM languages and tools available today offer only one hardwired style which cannot be adapted for different domains and circumstances. For example, Melanee and MetaDepth enforce rather strict modeling styles out-of-the-box, with no opportunity to change to alternative styles, while DMLA and MultiEcore offer more relaxed, but still fixed, default styles. On the other hand, although languages like DeepTelos and FMML^{x}, which are based on a small, meta-circular core, have default styles that can theoretically be changed, doing so is a highly technical task that requires expertise about the inner works of the supporting tools. By supporting full introspection capabilities, in both the linguistic and ontological dimensions, through a small and intuitive set of extensions to OCL, DOCL is one of the first deep constraint languages providing a simple and concise approach to adapting the applied MLM modeling style.

P2: The second problem of ambiguous modeling styles is a symptom of the fact that the (usually fixed) modeling styles supported by existing MLM languages and tools are either vaguely defined in natural language, or defined in a highly formal way using mathematical notations. As a result, the semantics and rules of the modeling style in place are often ambiguous and poorly understood by mainstream modelers. By supporting the definition of level spanning and introspecting constraints in a user-friendly, conservative extension of the OCL, DOCL facilitates the documentation of styles in a widely understandable and unambiguous way. This applies not only to optional styles of the kind described in Chapter 9, but also to the core and default styles described in Chapter 8. The descriptions of the core and default styles presented in those chapters provide the clearest definitions of those styles available to date.

P3: The third problem of unenforced modeling patterns is similar to the previous problem, but at a smaller level of granularity. While styles apply to one or more complete levels, patterns apply to small groups of clabjects. The most iconic pattern in MLM is the "powertype" pattern, since MLM's attempts to model this pattern in a cleaner and more intuitive way arguably stimulating the original development of MLM approaches. However, although languages like LML allow occurrences of the powertype pattern to be modeled in a much more natural way than traditional two-level modeling languages, the correct application of the involved relationships has to date been left to the modelers. Even if modelers are aware of the semantics of the pattern, and know how to apply it, the Melanee tool does not check that the rules of the pattern are actually adhered to as a model evolves. The level-scoping and introspection features of DOCL help address this problem by supporting precise and user-friendly definitions of patterns, in the context of specific groups of clabjects in a model, and by allowing adherence to the pattern's rules to be checked whenever the model is changed.

16.2 Validity of the Hypothesis

As described in Chapter 1, the hypothesis this thesis set out to validate is the following –

Hypothesis. "It is feasible to define and implement an OCL-based, deep constraint language to support reflective, level-adjuvant, and dimension-aware constraints on deep (i.e., multi-level) models represented using the LML".

Since DOCL, and its supporting prototype implementation, fulfill all the defined requirements, the hypothesis formulated at the beginning of this thesis is true. Its validity was confirmed systematically using the design science methodology in the following way -

- *Problem Relevance*: The relevance of a deep object constraint language with the specified features of multi-level awareness and introspection was confirmed by reviewing the literature as well as by modeling four high-profile "Challenges" defined by the Multi workshop community.

- *Design as an Artefact*: The proposed solution to the problem was reified through two main artifacts – (a) the DOCL language specification, including formal definitions of its syntax and semantics, and (b), the prototype DOCL implementation as a plugin to the Melanee MLM platform.
- *Design Evaluation*: We evaluated the utility, quality, and efficacy of the developed technology by using it to (a) precisely define, and enforce the use of, well-known multi-level modeling styles and (b) ensure the consistent use of well-known patterns in multi-level modeling.
- *Research Contributions*: The efficacy and utility of the developed artifacts were demonstrated and evaluated by using them in solutions to the four high-profile modeling challenges mentioned above. These are intended to provide "benchmark" modeling problems, for a range of domains, to exercise the full range of MLM capabilities. The LML solutions to these challenges include multiple DOCL constraints and could not have been modeled without DOCL's deep modeling features.
- *Research Rigor*: The properties of the developed artifacts, including the syntax and semantics, were defined formally using a grammar and first-order logic expressions, using OCL and DOCL itself.
- *Design as a Search Process*: The features of DOCL were refined in a feedback-oriented manner by exploring their utility in numerous scenarios, including the four benchmark challenges, and adapting them according to the lessons learned.
- *Communication of Research*: The work related to this thesis has been published in six workshop papers (in the MULTI workshop series), one conference paper, and one journal paper.

16.3 Future Work

There are many ways the artifacts developed in this (i.e., DOCL and its implementation) could be improved and further exploited. Three of the most interesting and potentially useful are described below.

16.3.1 Multiple Dimension Awareness

The current features of DOCL are optimized for the “classic” version of the OCA, which recognizes only two dimensions - a linguistic one and an ontological one. However, MLM languages that allow multiple ontological dimensions have already been proposed in the literature [79]. If these prove useful, they would benefit from enhanced versions of DOCL that provide a unified navigation and designation approach to all possible dimensions (i.e., multiple ontological and/or linguistic or meta-dimensions). This version would be “aware” of the existence of multiple classification levels and would provide features to support their exploitation. Generalizing the OCA to multiple dimensions, with appropriate multi-dimensional constraint support, could help further unify the many heterogeneous MLM languages currently available.

16.3.2 Smell Detection in MLM

In the context of modeling, “smells” refer to certain patterns or configurations within the model that might indicate design issues, inconsistencies, or potential problems. However, unlike anti-patterns, specific occurrences of them may be acceptable or even beneficial. These smells can be expressed as rules and constraints in DOCL, allowing modelers to define what constitutes a smell in the context of their specific domain.

The DOCL platform can therefore also provide a platform to detect model smells (like code smells). DOCL’s ability to navigate the models completely (even instances) and to access both linguistic and ontological meta-information make it particularly powerful for this purpose. Dedicated tools for detecting model smells already exist (e.g., checkstyle, decor, etc.), but these are not aware of MLM smells. DOCL could be fairly easily enhanced to characterize MLM smells and then to check for their presence through a deep model while evaluating the involved constraints at all levels.

16.3.3 Constraint Language of a Common Core

Ultimately, DOCL has the right mix of properties to serve as the constraint language for a common core language for MLM which builds on the core style defined in Section 8.2. As a conservative extension of the most well-known and widely-used constraint language in mainstream models (OCL) and a level-agnostic language founded on a minimalistic metamodel for deep modeling (LML), DOCL can easily be used to support a wide range of future styles and patterns. For example, DOCL was recently used as the constraint language in a paper [82] that explored field (i.e., attribute/slots) designs in MLM languages.

Bibliography

[1] *10th International Workshop on Multi-Level Modelling*. MULTI 2023. URL: https://jku-win-dke.github.io/MULTI2023/.

[2] *5th International Workshop on Multi-Level Modelling*. MULTI 2018. URL: https://www.wi-inf.uni-duisburg-essen.de/MULTI2018/index.html.

[3] *About the MOF Query/View/Transformation Specification Version 1.3*. URL: https://www.omg.org/spec/QVT/About-QVT/.

[4] *About the Unified Modeling Language Specification Version 2.5.1*. URL: https://www.omg.org/spec/UML/2.5.1/About-UML/.

[5] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. USA: Prentice-Hall, Inc., 1972. ISBN: 978-0-13-914556-8.

[6] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 978-0-321-48681-3.

[7] Sinan Si Alhir. *Guide to Applying the UML*. Springer Science & Business Media, 2006.

[8] Joao Paulo A. Almeida et al. “The MULTI Process Challenge”. In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. Munich, Germany: IEEE, Sept. 2019, pp. 164–167. ISBN: 978-1-72815-125-0. DOI: 10.1109/MODELS-C.2019.00027.

[9] Timo Asikainen and Tomi Männistö. “Nivel: A Metamodelling Language with a Formal Semantics”. In: *Software & Systems Modeling* 8.4 (Sept. 2009), pp. 521–549. ISSN: 1619-1374. DOI: 10.1007/s10270-008-0103-2.

[10] Timo Asikainen, Tomi Männistö, and Eetu Huovila. *nivel2: A web-based multi-level modelling environment built on a relational database*. 2023. DOI: 10.48550/arXiv.2303.12171.

[11] C. Atkinson and T. Kühne. “Model-Driven Development: A Metamodeling Foundation”. In: *IEEE Software* 20.5 (Sept. 2003), pp. 36–41. ISSN: 1937-4194. DOI: 10.1109/MS.2003.1231149.

[12] Colin Atkinson. “Meta-modelling for distributed object environments”. In: *Proceedings First International Enterprise Distributed Object Computing Workshop*. IEEE. 1997, pp. 90–101.

[13] Colin Atkinson and Ralph Gerbig. “Level-Agnostic Designation of Model Elements”. In: *Modelling Foundations and Applications*. Ed. by David Hutchison et al. Vol. 8569. Cham: Springer International Publishing, 2014, pp. 18–34. ISBN: 978-3-319-09194-5 978-3-319-09195-2. DOI: 10.1007/978-3-319-09195-2_2.

[14] 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. Innsbruck, Austria: Association for Computing Machinery, Sept. 2012, pp. 1–2. ISBN: 978-1-4503-1853-2.

[15] Colin Atkinson, Ralph Gerbig, and Bastian Kennel. “On-the-Fly Emendation of Multi-level Models”. In: *Modelling Foundations and Applications*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, July 2012, pp. 194–209. ISBN: 978-3-642-31490-2 978-3-642-31491-9. DOI: 10.1007/978-3-642-31491-9_16.

[16] Colin Atkinson, Ralph Gerbig, and Thomas Kühne. “Comparing Multi-Level Modeling Approaches”. In: *CEUR Workshop Proceedings*. Ed. by Colin Atkinson. Vol. 1286. Aachen, Germany: RWTH Aachen, 2014, pp. 53–61.

[17] Colin Atkinson, Matthias Gutheil, and Bastian Kennel. “A Flexible Infrastructure for Multilevel Language Engineering”. In: *IEEE Transactions on Software Engineering* 35.6 (Nov. 2009), pp. 742–755. ISSN: 1939-3520. DOI: 10.1109/TSE.2009.31.

[18] Colin Atkinson and Thomas Kühne. “Processes and Products in a Multi-Level Metamodeling Architecture”. In: *International Journal of Software Engineering and Knowledge Engineering* 11.06 (Dec. 2001), pp. 761–783. ISSN: 0218-1940, 1793-6403. DOI: 10.1142/S0218194001000724.

[19] Colin Atkinson and Thomas Kühne. “The Essence of Multilevel Metamodeling”. In: *UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. Ed. by Martin Gogolla and Cris Kobryn.

Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 19–33. ISBN: 978-3-540-45441-0.

[20] Colin Atkinson and Thomas Kühne. "In Defence of Deep Modelling". In: *Information and Software Technology* 64 (Aug. 1, 2015), pp. 36–51. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2015.03.010. URL: https://www.sciencedirect.com/science/article/pii/S0950584915000671.

[21] Sándor Bácsi et al. "Melanee and DMLA – A Contribution to the MULTI 2021 Collaborative Comparison Challenge". In: *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. Oct. 2021, pp. 556–565. DOI: 10.1109/MODELS-C53483.2021.00086.

[22] Paul Baker et al. *Model-Driven Testing: Using the UML Testing Profile*. Springer Science & Business Media, 2007.

[23] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. "Reasoning on UML Class Diagrams". In: *Artificial Intelligence* 168.1-2 (Oct. 2005), pp. 70–118. ISSN: 00043702. DOI: 10.1016/j.artint.2005.05.003.

[24] Jean Bézivin. "On the Unification Power of Models". In: *Software & Systems Modeling* 4.2 (May 2005), pp. 171–188. ISSN: 1619-1374. DOI: 10.1007/s10270-005-0079-0.

[25] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 2005. ISBN: 978-0-321-26797-9.

[26] Jordi Cabot and Martin Gogolla. "Object Constraint Language (OCL): A Definitive Guide". In: *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*. Ed. by Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 58–90. ISBN: 978-3-642-30982-3. DOI: 10.1007/978-3-642-30982-3_3.

[27] Luca Cardelli. "Structural subtyping and the notion of power type". In: POPL '88 (1988), 70–79. DOI: 10.1145/73560.73566.

[28] Rudolf Carnap. *Meaning and Necessity: A Study in Semantics and Modal Logic*. Ed. by 2d edition. Chicago, IL: University of Chicago Press, Feb. 1988. ISBN: 978-0-226-09347-5.

[29] Victorio A. Carvalho, João Paulo A. Almeida, and Giancarlo Guizzardi. “Using a Well-Founded Multi-level Theory to Support the Analysis and Representation of the Powertype Pattern in Conceptual Modeling”. In: *Advanced Information Systems Engineering*. Ed. by Selmin Nurcan et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 309–324. ISBN: 978-3-319-39696-5. DOI: 10.1007/978-3-319-39696-5_19.

[30] Victorio A. Carvalho and João Paulo A. Almeida. “Toward a Well-Founded Theory for Multi-Level Conceptual Modeling”. In: *Software & Systems Modeling* 17.1 (Feb. 1, 2018), pp. 205–231. ISSN: 1619-1374. DOI: 10.1007/s10270-016-0538-9.

[31] Victorio A. Carvalho et al. “Extending the Foundations of Ontology-Based Conceptual Modeling with a Multi-level Theory”. In: *Conceptual Modeling*. Ed. by Paul Johannesson et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 119–133. ISBN: 978-3-319-25264-3. DOI: 10.1007/978-3-319-25264-3_9.

[32] Michele Chinosi and Alberto Trombetta. “BPMN: An Introduction to the Standard”. In: *Computer Standards & Interfaces* 34.1 (2012), pp. 124–134.

[33] Noam Chomsky. “On the Notion “Rule of Grammar””. In: *Proceedings of Symposia in Applied Mathematics*. Ed. by Roman Jakobson. Vol. 12. Providence, Rhode Island: American Mathematical Society, 1961, pp. 6–24. ISBN: 978-0-8218-1312-6 978-0-8218-9227-5. DOI: 10.1090/psapm/012/9985.

[34] Noam Chomsky. *Syntactic Structures*. De Gruyter Mouton, Sept. 2009. ISBN: 978-3-11-021832-9. DOI: 10.1515/9783110218329.

[35] Tony Clark. “A Meta-Circular Basis for Model-Based Language Engineering.” In: *The Journal of Object Technology* 19.3 (2020), 3:1. ISSN: 1660-1769. DOI: 10.5381/jot.2020.19.3.a11.

[36] Tony Clark, Paul Sammut, and James Willans. *Applied Metamodelling: A Foundation for Language Driven Development (Third Edition)*. 2015.

[37] Tony Clark, Paul Sammut, and James Willans. *Superlanguages: Developing Languages and Applications with XMF*. Jan. 1, 2008.

[38] Tony Clark and James S. Willans. "Software Language Engineering with XMF and XModeler". In: *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. 2013, p. 30. ISBN: 978-1-4666-2092-6.

[39] Pierre Cointe. "Metaclasses Are First Class: The ObjVlisp Model". In: *SIGPLAN Not.* 22.12 (1987), 156–162. ISSN: 0362-1340. DOI: 10.1145/38807.38822.

[40] Vittorio Cortellessa and Antonio Pompei. "Towards a UML Profile for QoS: A Contribution in the Reliability Domain". In: *ACM SIGSOFT Software Engineering Notes* 29.1 (2004), pp. 197–206.

[41] Juan De Lara and Esther Guerra. "Deep Meta-modelling with MetaDepth". In: *Objects, Models, Components, Patterns*. Ed. by Jan Vitek. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–20. ISBN: 978-3-642-13953-6.

[42] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. "Model-Driven Engineering with Domain-Specific Meta-Modelling Languages". In: *Software & Systems Modeling* 14.1 (Feb. 2015), pp. 429–459. ISSN: 1619-1374. DOI: 10.1007/s10270-013-0367-z.

[43] Juan De Lara, Esther Guerra, and Jesús Sánchez Cuadrado. "When and How to Use Multilevel Modelling". In: *ACM Transactions on Software Engineering and Methodology* 24.2 (Dec. 2014), pp. 1–46. ISSN: 1049331X. DOI: 10.1145/2685615.

[44] Juan de Lara et al. "Extending Deep Meta-Modelling for Practical Model-Driven Engineering". In: *The Computer Journal* 57.1 (Jan. 2014), pp. 36–58. ISSN: 0010-4620, 1460-2067. DOI: 10.1093/comjnl/bxs144.

[45] Birgit Demuth. "The Dresden OCL toolkit and its role in Information Systems development". In: *Proc. of the 13th International Conference on Information Systems Development (ISD'2004)*. Vol. 7. 2004.

[46] Birgit Demuth and Claas Wilke. "Model and Object Verification by Using Dresden OCL". In: *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, Ufa, Russia*. 2009, pp. 687–690.

[47] Khanh-Hoang Doan and Martin Gogolla. "Extending a UML and OCL Tool for Meta-Modeling: Applications towards Model Quality Assessment". In: *Modellierung 2018*. Gesellschaft für Informatik e.V., 2018.

[48] Dirk Draheim. “Reflective Constraint Writing”. In: *Transactions on Large-Scale Data- and Knowledge-Centered Systems XXIV*. Ed. by Abdelkader Hameurlain et al. Vol. 9510. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 1–60. ISBN: 978-3-662-49213-0 978-3-662-49214-7. DOI: 10.1007/978-3-662-49214-7_1.

[49] Robert France and Bernhard Rumpe. “Model-Driven Development of Complex Software: A Research Roadmap”. In: *Future of Software Engineering (FOSE '07)*. May 2007, pp. 37–54. DOI: 10.1109/FOSE.2007.14.

[50] Ulrich Frank. “Multilevel Modeling. Toward a New Paradigm of Conceptual Modeling and Information Systems Design”. In: *Business & Information Systems Engineering* 6 (Dec. 2014), pp. 319–337. DOI: 10.1007/s12599-014-0350-4.

[51] Lidia Fuentes-Fernández and Antonio Vallecillo-Moreno. “An Introduction to UML Profiles”. In: *UML and Model Engineering* 2.6-13 (2004), p. 72.

[52] Erich Gamma et al. “Design Patterns: Abstraction and Reuse of Object-Oriented Design”. In: *ECOOP' 93 — Object-Oriented Programming*. Ed. by Oscar M. Nierstrasz. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1993, pp. 406–431. ISBN: 978-3-540-47910-9. DOI: 10.1007/3-540-47910-4_21.

[53] Ralph Gerbig. “Deep, Seamless, Multi-format, Multi-notation Definition and Use of Domain-specific Languages”. PhD thesis. Mannheim: University Mannheim, 2017.

[54] Seymour Ginsburg. *The Mathematical Theory of Context Free Languages*. McGraw-Hill, 1966. ISBN: 978-0-07-023280-8.

[55] Martin Gogolla and Antonio Vallecillo. “On Softening OCL Invariants.” In: *The Journal of Object Technology* 18.2 (2019), 6:1. ISSN: 1660-1769. DOI: 10.5381/jot.2019.18.2.a6.

[56] Cesar Gonzalez-Perez and Brian Henderson-Sellers. “A Powertype-Based Metamodelling Framework”. In: *Software & Systems Modeling* 5.1 (Apr. 1, 2006), pp. 72–90. ISSN: 1619-1374. DOI: 10.1007/s10270-005-0099-9.

[57] Georg Gottlob. “Adventures with Datalog: Walking the Thin Line Between Theory and Practice”. In: *AIxIA 2022 – Advances in Artificial Intelligence*. Ed. by Agostino Dovier, Angelo Montanari, and Andrea

Orlandini. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2023, pp. 489–500. DOI: 10.1007/978-3-031-27181-6_34.

[58] Matthew Hause et al. "The SysML Modelling Language". In: *Fifteenth European Systems Engineering Conference*. Vol. 9. 2006, pp. 1–12.

[59] Charlotte Herzeel, Pascal Costanza, and Theo D'Hondt. "Reflection for the Masses". In: *Self-Sustaining Systems*. Ed. by Robert Hirschfeld and Kim Rose. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 87–122. ISBN: 978-3-540-89275-5. DOI: 10.1007/978-3-540-89275-5_6.

[60] Alan R. Hevner et al. "Design Science in Information Systems Research". In: *MIS Quarterly* 28.1 (2004), pp. 75–105. ISSN: 0276-7783. DOI: 10.2307/25148625.

[61] John E. Hopcroft. *Introduction to Automata Theory, Languages, and Computation*. Pearson Education, Sept. 2008. ISBN: 978-81-317-2047-9.

[62] Walter L. Hürsch. "Should Superclasses Be Abstract?" In: *Proceedings of the 8th European Conference on Object-Oriented Programming*. ECOOP '94. London, UK, UK: Springer-Verlag, 1994, pp. 12–31. ISBN: 978-3-540-58202-1.

[63] Muzaffar Igamberdiev, Georg Grossmann, and Markus Stumptner. "A Feature-Based Categorization of Multi-Level Modeling Approaches and Tools". In: *CEUR Workshop Proceedings*. Vol. 1722. School of Information Technology and Mathematical Sciences. Germany Ruzica Piskac, 2016, pp. 35–44.

[64] *Information technology – Programming languages – Prolog*. Standard. 1995. URL: https://www.iso.org/standard/21413.html.

[65] Stanislaw Jarzabek and Tomasz Krawczyk. "LL-regular Grammars". In: *Information Processing Letters* 4.2 (Nov. 1975), pp. 31–37. ISSN: 0020-0190. DOI: 10.1016/0020-0190(75)90009-5.

[66] *Java Software*. Nov. 29, 2022. URL: https://www.oracle.com/java/.

[67] Rick Jelliffe. "The Schematron: An XML Structure Validation Language Using Patterns in Trees". In: 57 (2001).

[68] Manfred Jeusfeld, Gergely Mezei, and Sándor Bácsi. "DeepTelos and DMLA – A Contribution to the MULTI 2022 Collaborative Comparison Challenge". In: (2022), p. 11.

[69] Manfred A. Jeusfeld. *ConceptBase.Cc User Manual - Version 8.2*. URL: https://conceptbase.sourceforge.net/userManual82/CB-Manual.pdf.

[70] Manfred A. Jeusfeld and Bernd Neumayr. "DeepTelos: Multi-level Modeling with Most General Instances". In: *Conceptual Modeling*. Ed. by Isabelle Comyn-Wattiau et al. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 198–211. ISBN: 978-3-319-46397-1.

[71] Dominik Kantner. "Specification and Implementation of a Deep OCL Dialect". MA thesis. Mannheim, 2014.

[72] Bastian Kennel. "A Unified Framework for Multi-Level Modeling". PhD thesis. Mannheim: University Mannheim, 2012.

[73] Andrei Kirshin, Dolev Dotan, and Alan Hartman. "A UML Simulator Based on a Generic Model Execution Engine". In: *Models in Software Engineering*. Ed. by Thomas Kühne. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 324–326. ISBN: 978-3-540-69489-2. DOI: 10.1007/978-3-540-69489-2_40.

[74] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0-321-19442-X.

[75] Donald E. Knuth. "On the Translation of Languages from Left to Right". In: *Information and Control* 8.6 (Dec. 1965), pp. 607–639. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(65)90426-2.

[76] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. "The Epsilon Object Language (EOL)". In: *Model Driven Architecture – Foundations and Applications*. Ed. by Arend Rensink and Jos Warmer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 128–142. ISBN: 978-3-540-35910-4. DOI: 10.1007/11787044_11.

[77] Thomas Kühne. "Exploring Potency". In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS '18. New York, NY, USA: ACM, 2018, pp. 2–12. ISBN: 978-1-4503-4949-9. DOI: 10.1145/3239372.3239411.

[78] Thomas Kühne. "Multi-Dimensional Multi-Level Modeling". In: *Software and Systems Modeling* (Jan. 2022). ISSN: 1619-1374. DOI: 10.1007/s10270-021-00951-5.

[79] Thomas Kühne and Arne Lange. “Melanee and DLM – A Contribution to the MULTI Collaborative Comparison Challenge”. In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS ’22. Montreal, Quebec, Canada: Association for Computing Machinery, Oct. 2022, pp. 434–443. DOI: 10.1145/3550356.3561571.

[80] Thomas Kühne and Manfred Jeusfeld. *MULTI Warehouse Challenge*. 2023.

[81] Thomas Kühne and Friedrich Steimann. “Tiefe Charakterisierung”. In: *Modellierung 2004*. Bonn: Gesellschaft für Informatik e.V., 2004, pp. 109–119. ISBN: 3-88579-374-1.

[82] Thomas Kühne et al. “Field Types for Deep Characterization in Multi-Level Modeling”. In: *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2023, pp. 639–648. DOI: 10.1109/MODELS-C59198.2023.00105.

[83] Arne Lange and Colin Atkinson. “Multi-Level Modeling with LML: A Contribution to the Multi-Level Process Challenge”. In: *Enterprise Modelling and Information Systems Architectures (EMISAJ)* 17 (June 2022), 6:1–36. ISSN: 1866-3621. DOI: 10.18417/emisa.17.6.

[84] Arne Lange and Colin Atkinson. “Multi-Level Modeling with MELANEE”. In: *CEUR Workshop Proceedings*. Aachen: RWTH, 2018, pp. 653–662.

[85] Arne Lange and Colin Atkinson. “On the Rules for Inheritance in LML”. In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. Sept. 2019, pp. 113–118. DOI: 10.1109/MODELS-C.2019.00021.

[86] P. M. Lewis and R. E. Stearns. “Syntax-Directed Transduction”. In: *Journal of the ACM* 15.3 (July 1968), pp. 465–488. ISSN: 0004-5411. DOI: 10.1145/321466.321477.

[87] Barbara H. Liskov and Jeannette M. Wing. “A Behavioral Notion of Subtyping”. In: *ACM Transactions on Programming Languages and Systems* 16.6 (Nov. 1994), pp. 1811–1841. ISSN: 0164-0925. DOI: 10.1145/197320.197383.

[88] Shourong Lu, Wolfgang A Halang, and Lichen Zhang. “A Component-Based UML Profile to Model Embedded Real-Time Systems Designed by the MDA Approach”. In: *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA’05)*. IEEE, 2005, pp. 563–566.

[89] Fernando Macias Gomez de Villar, Adrian Rutle, and Volker Stolz. “MultEcore: Combining the Best of Fixed-Level and Multilevel Metamodelling”. In: (2016). ISSN: 1613-0073.

[90] Pattie Maes. “Concepts and Experiments in Computational Reflection”. In: *ACM SIGPLAN Notices* 22.12 (Dec. 1, 1987), pp. 147–155. ISSN: 0362-1340. DOI: 10.1145/38807.38821.

[91] Bertrand Meyer. “Applying’design by Contract’”. In: *Computer* 25.10 (1992), pp. 40–51.

[92] Gergely Mezei et al. “The MULTI Collaborative Comparison Challenge”. In: *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. Fukuoka, Japan: IEEE, Oct. 2021, pp. 495–496. ISBN: 978-1-66542-484-4. DOI: 10.1109/MODELS-C53483.2021.00077.

[93] John Mylopoulos. “Conceptual Modelling and Telos”. In: *Conceptual modelling, databases, and CASE: An integrated view of information system development* (1992).

[94] Anton Nijholt. *Context-Free Grammars: Covers, Normal Forms, and Parsing*. Springer, 1980. ISBN: 978-0-387-10245-0. DOI: 10.1007/3-540-10245-0.

[95] James Odell. “Power Types”. In: *Journal of Object-Oriented Programming* 7.2 (1994), pp. 8–12.

[96] OMG. *About the Object Constraint Language Specification Version 2.4*. https://www.omg.org/spec/OCL/2.4/PDF.

[97] Xavier Oriol and Ernest Teniente. “OCL$$_\textsf {UNIV}$$: Expressive UML/OCL Conceptual Schemas for Finite Reasoning”. In: *Conceptual Modeling*. Ed. by Heinrich C. Mayr et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 354–369. ISBN: 978-3-319-69904-2. DOI: 10.1007/978-3-319-69904-2_28.

[98] Xavier Oriol, Ernest Teniente, and Albert Tort. “Computing Repairs for Constraint Violations in UML/OCL Conceptual Schemas”. In: *Data & Knowledge Engineering*. Selected Papers from the 33rd International Conference on Conceptual Modeling (ER 2014) 99 (Sept. 1, 2015), pp. 39–58. ISSN: 0169-023X. DOI: 10.1016/j.datak.2015.06.006.

[99] Terence Parr, Sam Harwell, and Kathleen Fisher. “Adaptive LL(*) Parsing: The Power of Dynamic Analysis”. In: *ACM SIGPLAN Notices* 49.10 (Oct. 2014), pp. 579–598. ISSN: 0362-1340. DOI: 10.1145/2714064.2660202.

[100] Guillermo Polito et al. “A Bootstrapping Infrastructure to Build and Extend Pharo-like Languages”. In: *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* Onward! 2015. New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 183–196. ISBN: 978-1-4503-3688-8. DOI: 10.1145/2814228.2814236.

[101] John Poole et al. *Common Warehouse Metamodel*. John Wiley & Sons, 2002.

[102] Franklin Ramalho, Jacques Robin, and Roberto Barros. “XOCL - an XML Language for Specifying Logical Constraints in Object Oriented Models”. In: *JUCS - Journal of Universal Computer Science* 9.8 (Aug. 2003), pp. 956–969. ISSN: 0948-6968. DOI: 10.3217/jucs-009-08-0956.

[103] Mark Richters and Martin Gogolla. “On Formalizing the UML Object Constraint Language OCL”. In: *Conceptual Modeling – ER ’98*. Ed. by Tok-Wang Ling, Sudha Ram, and Mong Li Lee. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1998, pp. 449–464. ISBN: 978-3-540-49524-6. DOI: 10.1007/978-3-540-49524-6_35.

[104] Mark Richters and Martin Gogolla. “Validating UML Models and OCL Constraints”. In: *UML 2000 — The Unified Modeling Language*. Ed. by Gerhard Goos et al. Vol. 1939. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 265–277. ISBN: 978-3-540-41133-8 978-3-540-40011-0. DOI: 10.1007/3-540-40011-7_19.

[105] Alejandro Rodríguez and Fernando Macías. “Multilevel Modelling with MultEcore: A Contribution to the MULTI Process Challenge”. In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. Sept. 2019, pp. 152–163. DOI: 10.1109/MODELS-C.2019.00026.

[106] Alejandro Rodríguez et al. "A Foundation for the Composition of Multilevel Domain-Specific Languages". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. Sept. 2019, pp. 88–97.

[107] D. J. Rosenkrantz and R. E. Stearns. "Properties of Deterministic Top down Grammars". In: *Proceedings of the First Annual ACM Symposium on Theory of Computing*. STOC '69. New York, NY, USA: Association for Computing Machinery, May 1969, pp. 165–180. ISBN: 978-1-4503-7478-1. DOI: 10.1145/800169.805431.

[108] Arto Salomaa. *Jewels of Formal Language Theory*. Computer Science Press, 1981. ISBN: 978-0-914894-69-8.

[109] D.C. Schmidt. "Guest Editor's Introduction: Model-Driven Engineering". In: *Computer* 39.2 (Feb. 2006), pp. 25–31. ISSN: 1558-0814. DOI: 10.1109/MC.2006.58.

[110] Brian Selic. "Introduction". In: *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*. Ed. by Tony Clark and Jos Warmer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002, pp. 1–3. ISBN: 978-3-540-45669-8. DOI: 10.1007/3-540-45669-4_1.

[111] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory: Languages and Parsing*. Springer-Verlag, 1988. ISBN: 978-0-387-13720-9.

[112] Herbert Stachowiak. *Allgemeine Modelltheorie*. Wien, New York: Springer Verlag, 1973.

[113] Friedrich Steimann, Robert Clarisó, and Martin Gogolla. "OCL Rebuilt, From the Ground Up". In: *ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Västerås, Oct. 2023.

[114] *The Community for Open Innovation and Collaboration | The Eclipse Foundation*. URL: https://www.eclipse.org/.

[115] *The Epsilon Book*. https://www.eclipse.org/epsilon/doc/book/.

[116] Frédéric Thomas et al. "MARTE : le futur standard OMG pour le développement dirigé par les modèles des systèmes embarqués temps réel". In: 2007. URL: https://api.semanticscholar.org/CorpusID:192877475.

[117] Dániel Urbán, Gergely Mezei, and Zoltán Theisz. "Formalism for Static Aspects of Dynamic Metamodeling". In: *Periodica Polytechnica Electrical Engineering and Computer Science* 61.1 (2017), pp. 34–47. ISSN: 2064-5279. DOI: 10.3311/PPee.9547.

[118] Dániel Urbán, Zoltán Theisz, and Gergely Mezei. "Self-Describing Operations for Multi-level Meta-modeling". In: *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*. MODELSWARD 2018. Setubal, PRT: SCITEPRESS - Science and Technology Publications, Lda, Jan. 2018, pp. 519–527. ISBN: 978-989-758-283-7. DOI: 10/jqpf.

[119] Jos B. Warmer. *The Object Constraint Language: Getting Your Models Ready for MDA*. 2nd edition. Boston, MA: Addison-Wesley, 2003. ISBN: 978-0-321-17936-4.

[120] Jos B. Warmer. *The Object Constraint Language: Precise Modeling with UML*. Object Techology Series. Reading, Mass. ; Bonn [u.a.]: Addison Wesley, 1999. ISBN: 978-0-201-37940-2.

[121] Eclipse Web. *Eclipse OCL™ (Object Constraint Language)*. Oct. 14, 2023. URL: https://projects.eclipse.org/projects/modeling.mdt.ocl.

[122] *XMI*. URL: http://www.omg.org/spec/XMI/.

Appendix A

DOCL Grammar in ANTLR 4

```
grammar DeepOcl;

contextDeclCS
:
  (
    propertyContextDeclCS
    | classifierContextCS
    | operationContextCS
  )+
;

operationContextCS
:
  CONTEXT levelSpecificationCS?
  (
    ID ':'
  )?
  (
    ID '::'
    (
      ID '::'
    )* ID
    | ID
  ) '('
  (
    parameterCS
    (
      ',' parameterCS
    )*
  )? ')'
  (
    ':' typeExpCS
  )?
  (
    preCS
    | postCS
    | bodyCS
  )*
;

levelSpecificationCS
:
  '(' NumberLiteralExpCS
  (
    ','
    (
      '_'
      | NumberLiteralExpCS
    )
  )? ')'
;

CONTEXT
:
  'context'
;

bodyCS
:
  'body' ID? ':' specificationCS
;

postCS
:
  'post' ID? ':' specificationCS
;

preCS
:
  'pre' ID? ':' specificationCS
;

defCS
:
  'def' ID? ':' ID
  (
    (
      '(' parameterCS?
      (
        ',' parameterCS
      )* ')'
    )? ':' typeExpCS? '=' specificationCS
  )
;

filterCS
:
    'filter' ID? ':' specificationCS
;

typeExpCS
:
  typeNameExpCS
  | typeLiteralCS
;

typeLiteralCS
:
  primitiveTypeCS
  | collectionTypeCS
  | tupleTypeCS
;
```

```
tupleTypeCS
:
  'Tuple'
  (
    '(' tuplePartCS
    (
      ',' tuplePartCS
    )* ')'
    | '<' tuplePartCS
    (
      ',' tuplePartCS
    )* '>'
  )?
;

tuplePartCS
:
  ID ':' typeExpCS
;

collectionTypeCS
:
  collectionTypeIDentifier
  (
    '(' typeExpCS ')'
    | '<' typeExpCS '>'
  )?
;

collectionTypeIDentifier
:
  'Collection'
  | 'Bag'
  | 'OrderedSet'
  | 'Sequence'
  | 'Set'
;

primitiveTypeCS
:
  'Boolean'
  | 'Integer'
  | 'Real'
  | 'ID'
  | 'UnlimitedNatural'
  | 'OclAny'
  | 'OclInvalID'
  | 'OclVoID'
;

typeNameExpCS
:
  ID '::'
  (
    ID '::'
  )* ID
  | ID
```

```
;

specificationCS
:
  infixedExpCS*
;

expCS
:
  infixedExpCS
;

infixedExpCS
:
  prefixedExpCS # prefixedExp
  | iteratorBarExpCS # iteratorBar
  | left = infixedExpCS op = '^' right = infixedExpCS # Message
  | left = infixedExpCS op = 'implies' right = infixedExpCS # implies
  | left = infixedExpCS op =
  (
    'xor'
    | 'or'
    | 'and'
  ) right = infixedExpCS # andOrXor
  | left = infixedExpCS op =
  (
    '='
    | '<>'
    | '<='
    | '>='
    | '<'
    | '>'
  ) right = infixedExpCS # equalOperations
  | left = infixedExpCS op =
  (
    '+'
    | '-'
  ) right = infixedExpCS # plusMinus
  | left = infixedExpCS op =
  (
    '*'
    | '/'
  ) right = infixedExpCS # timesDivide
;

iteratorBarExpCS
:
  '|'
;

navigationOperatorCS
:
  '.' # dot
  | '->' # arrow
;

prefixedExpCS
:
  UnaryOperatorCS+ primaryExpCS
  | primaryExpCS
  (
    navigationOperatorCS primaryExpCS
  )*
  | primaryExpCS
;

UnaryOperatorCS
:
  '-'
  | 'not'
;

primaryExpCS
:
  letExpCS
  | ifExpCS
  | navigatingExpCS
  | selfExpCS
  | primitiveLiteralExpCS
  | tupleLiteralExpCS
  | collectionLiteralExpCS
  | typeLiteralExpCS
  | nestedExpCS
;

nestedExpCS
:
  '(' expCS+ ')'
;

ifExpCS
:
  'if' ifexp = expCS+ 'then' thenexp = expCS+ 'else' elseexp = expCS+ 'endif'
;

letExpCS
:
  'let' letVariableCS
  (
    ',' letVariableCS
  )* 'in' in = expCS+
;

letVariableCS
:
  name = ID ':' type = typeExpCS '=' exp = expCS+
;

typeLiteralExpCS
:
  typeLiteralCS
;

collectionLiteralExpCS
:
  collectionTypeCS '{'
  (
    collectionLiteralPartCS
    (
      ',' collectionLiteralPartCS
    )*
  )? '}'
;

collectionLiteralPartCS
:
  expCS
  (
    '..' expCS
  )?
;

tupleLiteralExpCS
:
  'Tuple' '{' tupleLiteralPartCS
  (
    ',' tupleLiteralPartCS
  )* '}'
;

tupleLiteralPartCS
:
  ID
  (
    ':' typeExpCS
  )? '=' expCS
;

selfExpCS
:
  'self'
;

primitiveLiteralExpCS
:
  NumberLiteralExpCS # number
  | STRING # string
  | BooleanLiteralExpCS # boolean
  | InvalIDLiteralExpCS # invalid
  | NullLiteralExpCS # null
;

InvalIDLiteralExpCS
:
  'invalid'
;

NumberLiteralExpCS
:
  INT
  (
    '.' INT
  )?
  (
    (
      'e'
      | 'E'
    )
    (
      '+'
      | '-'
    )? INT
  )?
;

fragment
DIGIT
:
  [0-9]
;

INT
:
  DIGIT+
;

BooleanLiteralExpCS
:
  'true'
  | 'false'
;

NullLiteralExpCS
:
  'null'
;

navigatingExpCS
:
  opName = indexExpCS
  (
    '@' 'pre'
  )?
  (
    '(' '"'? onespace? arg = navigatingArgCS* commaArg = navigatingCommaArgCS*
    barArg = navigatingBarAgrsCS* semiArg = navigatingSemiAgrsCS* '"'? ')'
  )*
;

navigatingSemiAgrsCS
:
  ';' navigatingArgExpCS
  (
    ':' typeExpCS
  )?
  (
    '=' expCS+
  )?
;

navigatingCommaArgCS
:
  ',' navigatingArgExpCS
  (
    ':' typeExpCS
  )?
  (
    '=' expCS+
  )?
;

navigatingArgExpCS
:
  iteratorVariable = infixedExpCS iteratorBarExpCS nameExpCS
  navigationOperatorCS body = infixedExpCS*
  | infixedExpCS+
;

navigatingBarAgrsCS
:
  '|' navigatingArgExpCS
  (
    ':' typeExpCS
  )?
  (
    '=' expCS+
  )?
;

navigatingArgCS
:
  navigatingArgExpCS
  (
    ':' typeExpCS
  )?
  (
    '=' expCS+
  )?
;

indexExpCS
:
  nameExpCS
  (
    '[' expCS
    (
      ',' expCS
    )* ']'
  )?
;

nameExpCS
:
  (
    (
      ID '::'
      (
        ID '::'
      )* ID
    )
```

```
      | ID
      | STRING
    ) # name
    | '$' clab = ID '$' # ontologicalName
    | '#' aspect = ID
    (
      '('
      (
        NumberLiteralExpCS
        | ID
      )?
      (
        ','
        (
          NumberLiteralExpCS
          | ID
        )
      )* ')'
    )? '#' # linguisticalName
;

parameterCS
:
    (
      ID ':'
    )? typeExpCS
;

invCS
:
    'inv'
    (
      ID
      (
        '(' specificationCS ')'
      )?
    )? ':' specificationCS
;

classifierContextCS
:
   CONTEXT levelSpecificationCS?
    (
      ID ':'
    )?
    (
      (
        ID '::'
        (
          ID '::'
        )* ID
      )
      | ID
    )
    (
      invCS
      | defCS
    )*
```

```
;

propertyContextDeclCS
:
  CONTEXT levelSpecificationCS?
  (
    (
      ID '::'
      (
        ID '::'
      )* ID
    )
    | ID
  ) ':' typeExpCS
  (
    (
      initCS derCS?
    )?
    | derCS initCS?
  )
;

derCS
:
  'derive' ':' specificationCS
;

initCS
:
  'init' ':' specificationCS
;

ID
:
  [a-zA-Z] [a-zA-Z0-9]*
;

WS
:
  [ \t\n\r]+ -> skip
;

onespace
:
  ONESPACE
;

ONESPACE
:
  ' '
;

STRING
:
  '"'
  (
    ~[\r\n"]
    | '""'
  )* '"'
;

COMMENT
:
  '--' .*? '\n' -> skip
;
```

Appendix B

PLM Operation Reference

For the *DeepModel* meta-model the following methods and references are defined:

- `getContent()` – returns all containing elements; in this case all levels that are defined within the *DeepModel* instance
- `enumeration` – returns all the defined enumerations
- `getLevelAtIndex(int level)` – returns the level that is identified by the parameter
- `getPrimitiveDatatypes()` – returns all primitive data types
- `getAllDatatypes()` – returns all primitive data types and enumerations

The reference navigation can be identified by the missing parentheses at the end. For the *Level* meta-model the following operations and reference navigations are defined:

- `getContent()` – returns all containing elements of the *Level* instance.
- `getAllInheritances()` – returns all the generalizations that are present at the level
- `getClabjects()` –returns all elements that are of the type *Clabject* of the *Level*
- `getEntities` – returns all entities which are a subset of all *Clabjects* of the *Level*
- `getConnections()` – returns all *Connections* that are present at this *Level*
- `getClassifications()` – returns all classifications if the instance is present at this *Level*

- `getDeepModel` – returns the *DeepModel* that contains this *Level*
- `isRootLevel()` – returns true if the *Level* is the topmost level in the *DeepModel*, else false
- `isLeafLevel()` – returns true if the *Level* is the bottom level in the *DeepModel*, else false

Here the `content` navigation as well is referring to references of the meta-model.

The next list will display a selection of *Connection* operations and reference navigations, which are supposed to be useful for writing statements in the DeepOCL dialect efficiently.

- `getDomain()` – returns all destinations of the navigable connection ends of this *Connection*
- `getNotDomain()` –returns all *Clabjects* that participate in this *Connection* but are not navigable
- `getHumanReadableName()` – returns a human-readable name of this *Connection*
- `getParticipants()` – returns all participants, i.e. destinations of the connection ends, of this *Connection*
- `getMoniker()` – returns the moniker for this *Connection*
- `getMonikerForParticipant(Clabject)` – returns the moniker of this *Connection* for the parameter *Clabject* if it is reachable through this *Connection*
- `getOrder()` – returns the number of connection ends in the *Connection*
- `getParticipantForMoniker(String)` – returns the *Clabject* reachable through the *Connection* via the parameter *moniker*
- `getAllConnectionEnd` – returns the connection ends that the connection inherits from its supertypes

The operation `getAllConnectionEnd()` could also be replaced by the `connectionEnd` reference navigation.

Even though the *Clabject* meta-model contains many methods and references that can be invoked by any OCL statement, the following will only display a few operations and references which have a higher chance of being used when writing DeepOCL expressions.

- `getPotency()` – returns the potency of the *Clabject*
- `getContent()` – returns all the elements that are contained by the *Clabject*
- `getAllFeatures()` – returns all attached *Feature* entities, which could be from type *Attribute* or *Method*
- `getTypes()` – returns a collection of all *Clabjects* that are of the type of the source *Clabject*
- `getInstances` – returns all the *Clabjects* that are an instance of the source *Clabject* based on classification elements.
- `getAllAttributes()` – returns all *Attributes* of the source *Clabject*
- `getAllMethods()` – returns all the methods that are contained by the source *Clabject*
- `getDefinedNavigations()` – returns all defined navigation of the source *Clabject*
- `getDirectTypes()` – returns the direct types of the source *Clabject* based on the classification hierarchy
- `getDirectType()` – returns the direct type of a clabject
- `getDefinedInstances()` – returns the instances and their subtypes of the *Clabject* only
- `getSubtypes()` – returns all entities that inherit from the source *Clabject*
- `getSupertypes()` – returns the *Clabjects* this *Clabject* inherits properties from
- `getConnections()` – returns all connections from the source *Clabject*
- `getLevelIndex()` – returns the level index the source *Clabject* is located on
- `detDeepModel()` – returns the *DeepModel* the *Clabject* is contained in
- `isTypeOf(Clabject)` – returns true if the *Clabject* is in the classification tree of the *Clabject* that was passed in the parameter

For *Features*, attributes, and methods, we offer the following linguistic operations.

- `getClabject()` – return the clabject the feature is contained in
- `getDurability()` – returns the integer value of the durability of the feature

The linguistic operations that are specific to attributes are presented in the following (the *Attributes* class is a subclass of the *Feature* class)

- `getMutability()` – returns the mutability value of the attribute
- `getPossibleDataTypes()` – returns a list of all possible data types which are comprised of the primitive data types hard coded into Melanee and the user-defined enumeration types.
- `getPrimitiveDataTypes()` – returns a list of all hard-coded data types in Melanee
- `isEnumeration()` – returns true if the type of the attribute is defined by an enumeration
- `getEnumeration()` – returns a list of all user-defined enumerations in the model
- `getLiterals()` – returns a list of literals that are used in the enumeration

The linguistic operations that are specific to methods are presented in the following (the *Method* class is a subclass of the *Feature* class). As the parameters of each method have to be typed, like in JAVA, we can get the type information out of these parameters.

- `getInput()` – returns the list of input parameters of that method
- `getOutput` – returns the output parameter of that function
- `getBody()` – returns the string of the method body

For the *Inheritance* class, we offer the following linguistic dimension operations.

- `isDisjoint()` – returns true if the generalization set is disjoint and false if it is overlapping
- `isComplete()` – returns true if the generalization set is complete, i.e., is fully described, and false for incomplete

- `isIntersection()` – returns true if the generalization set is an intersection
- `getSuperTypes()` – returns a list of all the super clabjects of the inheritance.
- `getSubTypes()` – returns a list of all sub-clabjects in the inheritance.

These linguistic navigations are a vital part of navigating the deep model and keeping the model valid with respect to the constraints. Combined with the ontological navigation it shows the capabilities of the DeepOCL dialect.