A UNIFIED FRAMEWORK FOR MULTI-LEVEL MODELING

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

vorgelegt von
Bastian Kennel
aus Mannheim

Mannheim, 2012

Dekan:	Professor Dr. Heinz Jürgen Müller, Universität Mannheim
Referent:	Professor Dr. Colin Atkinson, Universität Mannheim
Korreferent:	Professor Dr. Uwe Aßmann, Universität Dresden

Tag der mündlichen Prüfung: 26. Juni 2012

Abstract

With the growing importance of modeling in software engineering and knowledge engineering, and the accelerating convergence of these two disciplines through the confluence of internet-based software applications, the need for a simple, unified information modeling framework fulfilling the use cases of both communities has increased significantly over recent years. These use cases include switching seamlessly between exploratory and constructive modes of modeling, representing all objects of relevance to a system using a precise engineering-oriented notation, and applying a wide range of automated checking and reasoning services to models to enhance their quality.

This thesis lays the foundation for such a framework by formalizing and extending the multi-level modeling paradigm developed by Atkinson & Kühne, building a practical prototype tool based on the widely-used Eclipse EMF toolkit. This paradigm represents the best foundation for such a framework because it can capture all objects of relevance to a system, at all levels of classification (e.g. instances, types, metatypes, metametatypes etc ...), in a uniform and extensible way regardless of when and how they came into existence. Multi-level models can therefore accomodate the generation and use of information from the exploration and discovery phases of a project right through to the operation and run-time execution phases, seamlessly changing the way the information is interpreted and processed as needed.

The developed framework and tool (Multi-level modeling and ontology engineering Environment, Melanie) encompasses all the typical ingredients of a model-driven development environment: a (meta) model (the Pan Level Model, PLM), a concrete syntax (The Level-agnostic Modeling Language, LML) and a formal semantics based on set theory. In addition, the framework supports the full range of model querying, checking and evolution services supported by standard software engineering and knowledge engineering tools. This includes full support for the constructive generation of instances from types and the exploratory discovery of new information based on existing model content (e.g. subsumption). To demonstrate the practical usability of the technology, the approach is applied to two well known examples from the software engineering and knowledge engineering communities – The Pizza ontology from the Protégé documentation and the Royal & Loyal example from the OCL documentation.

Zusammenfassung

Durch die wachsende Bedeutung von Modelierung sowohl in der Softwareentwicklung als auch Knowledge Engineering wuchs der Bedarf an einem einfachen, einheitlichen Rahmenwerk welches beide Anwendungsfälle erfüllt erheblich. Diese Entwicklung wird verstärkt durch die zunehmende Verschmelzung der beiden Disziplinen und die Bedeutung von internetbasierten Anwendungen. Typische Anwendungsfälle aus beiden Domänen umfassen das Umschalten zwischen dem schöpferischen und erforschenden Modelierungsmodus, die Darstellung der relevanten Objekte eines Systems über den gesamten Lebenszyklus mittels einer geeigneten Syntax und einer präzisen Semantik, sowie die Anwendung von automatisierten Schlussfolgerungen.

Das vorgestellte Rahmenwerk und die zugehörige Anwendung (Multi-level modeling and ontology engineering Environment, Melanie) beinhalten die typischen Merkmale einer modelgetriebenen Softwareentwicklungsumgebung: Ein Metamodel (Pan Level Model, PLM), eine konkrete Syntax (Level-agnostic Modeling Language, LML) und eine auf Mengentheorie basierte formale Semantik. Zusätzlich unterstützt es die selben Modelanfrage-, Validierungs- und Evolutionsservices welche sich in Standardwerkzeugen der Softwareentwicklung bzw. Knowlegde Engineering wiederfinden. Dies beinhaltet die Erstellung von Instanzen

basierend auf Typdefinitionen sowie das Schlussfolgern von neuen Informationen aus dem bestehenden Modelinhalt (z.B. Subsumption). Um die praktische Relevanz des Ansatzes zu unterstreichen sind zwei Anwendungsbeispiele realisiert – Die Pizza Ontology aus der Protégé Dokumentation sowie das Royal & Loyal Beispiel aus der OCL Dokumentation.

To Nina

Acknowledgements

I would like to thank my supervisor Colin Atkinson. When I first joined his group in 2004 it was only for a tutor job, but I was immediately impressed by the gentle working atmosphere. Right from the beginning when I returned to study for my PhD in 2008, Colin was very closely involved in the activities evolving around this research topic. The discussions were always helpful and open minded, equal and passionate. In every research paper we submitted to either a journal or a conference, Colin was the main driver and contributed most of the text.

I also would like to thank two former students whom I had the pleasure of working with: Björn Goß and Ralph Gerbig. When Björn joined me the PLM was still in its very early stages. There was no complete implementation, no graphical output and no reasoning. Together with Björn I implemented the first version of the PLM together with an engine that could plot a PLM model as SVG code. Unfortunately we did not anticipate the benefits the Eclipse world and its modeling projects could offer, so our python source code was discontinued many years ago and will most likely stay that way. I am glad to pass the torch on to Björn when he returns from his Master studies in England to work out the reasoning part of the PLM. Ralphs diploma thesis really pushed the technology to the next level as he introduced us to the world of eclipse based frameworks with a knowledge we could

not dream of. Without his dedication and excellent knowledge in these technologies, Melanie would be nowhere near what it is today. If it weren't for Ralph, we would still be reinventing the wheel over and over again and would be struggling with all the setbacks that come with developing an application from scratch.

Contents

List of Figures

LIST OF FIGURES

List of Algorithms

LIST OF ALGORITHMS

List of Theorems

LIST OF THEOREMS

LIST OF THEOREMS

Glossary

artefact a property or a clabject

blueprint the type that has been used to create an the instance. That instance is therefore by definition

complete type a type of an instance that defines exactly the properties of that instance and no less. The instance is thus an isonym of that type

computed information is a piece of information that has not been input into the ontology by the user but has been created by an automated service

correlation a model element that makes a statement about the relationship between the sets of instances of two or more clabjects

EMF Eclipse Modeling Framework

expressed information a piece of information that has been explicitly input into the ontology by the user

hyponym an instance of a type that has more properties than required by the type

incomplete type a type of an instances that defines less properties than those possessed by the instance. The instance is this a hyponym of that type

isonym an instance of a type that has only the necessary properties, and no more

KE Knowledge Engineering

LML Level Agnostic Modeling Language

Melanie Multi-Level Modeling And Ontology Engineering Environment

MLM Multi-Level modeling

MOF Meta Object Facility

OCA Orthografic Classification Architecture

OCL Object Constraint Language

offspring an instance that was creating by instantiating the type. It is by definition an isonym of that type

OMG Object Management Group

OWL Web Ontology Language

PLM Pan Level Model

PMLM Potency based Multi-Level Modeling

primary information a piece of information that is assumed to be correct by definition. Its validity is beyond doubt.

RDFS Resource Description Framework Schema

SE Software Engineering

secondary information a piece of information whose correctness is not beyond doubt and may be revised

trait a linguistic building attribute of a model element in an ontology.

UML Unified Modeling Language

XML Extensible Markup Language

Glossary

Chapter 1

Introduction

Visual modeling, in various guises, is now an indispensable part of virtually all modern IT projects, whether it be to visualize requirements, architectures and designs in software engineering, describe "ontologies" in knowledge engineering, specify data types and relationships in database development or represent processes in business automation. The quality of models, and the ease by which human users can create and maintain them, therefore has a major impact on the success of today's IT projects.

1.1 Observed Weaknesses

Despite significant advances in the state-of-the-art in modeling over recent years, the current generation of modeling languages and tools still have some fundamental weaknesses that often make models more complex than they need be, and thus more difficult to create, maintain and understand than necessary.

Fundamental Weakness 1 (Fragmentation)**:** One fundamental weakness is the traditional fragmentation of modeling technologies into two main blocks reflecting the two major traditions and user communities in which modeling evolved. One is the so called "software engineering" tradition of modeling

which evolved for the purpose of describing the properties and components of software systems with a view to supporting their construction. This includes the family of "entity relationship" oriented modeling technologies (26, 27) (focussing on the construction of information systems driven by relational databases) and the family of modeling technologies centred on the Unified Modeling Language (UML)(51, 56) (focusing on the developing of software systems using third generation, object-oriented programming languages). We refer to this tradition as the "constructive modeling" tradition. The other is the so called "knowledge engineering" tradition which evolved for the purpose of capturing the set theoretic properties of subjects of interest with a view to supporting computationally efficient reasoning (i.e. inference) operations underpinning "artificial intelligence". Today, this mainly includes the family of ontology and metadata representation language revolving around semantic web technologies, such as the Resource Description Framework Schema (RDFS) (72) and Web Ontology Language (OWL) (1, 18, 49). Since the capturing of knowledge often involves an exploration of the ideas and facts occurring in a given subject of interest, we refer to this tradition as the "exploratory modeling" tradition.

Fundamental Weakness 2 (Assumptions & Forced Choices)**:** Although the visual models developed in the two blocks are essentially made up of the same conceptual ingredients (i.e. classes, instances, attributes, relationships,...), the languages used to represent them are founded on different underlying assumptions and concrete semantic choices which makes their seamless couse impossible (7). This was not a problem when the two user communities were essentially separate and there was little interchange between them. However, with the growing importance of the Internet and the semantic web in almost all domains of computing, software engineers are increasingly having to deal with visual models from both traditions when developing software systems. Instead of focussing exclusively on the essence of the problem in hand, software engineers have to waste significant time worrying about the idiosyncrasies of the model from the two traditions and working out which of the mix of supported features are available and useful at what

time.

Fundamental Weakness 3 (Concrete Syntax)**:** In fact, Knowledge Engineering (KE) and Software Engineering (SE) technologies for modeling also have opposite strengths and weaknesses. The UML, the main language used in the SE tradition of modeling, has powerful visualization features (i.e. concrete syntax and visualization metaphors) but has not been placed on a solid, formal foundation with well understood semantics. In contrast, OWL, the main language used in the KE tradition of modeling, has a strong, solid formal semantic foundation (based on description logic) but very human unfriendly native syntax (Extensible Markup Language (XML) (21)) and visualization conventions (e.g. Protégé bubbles (40, 43)). Many ontology engineers therefore already informally used UML diagramming notations to visualize OWL ontologies.

Fundamental Weakness 4 (Linear Modeling Architecture)**:** Another fundamental weakness is that visual models from both modeling traditions are rooted in an approach to language definition and use that evolved from the hard-wired, text-oriented languages that dominated the previous millennium. Programming environments from this era typically relied on languages with a frozen syntax (abstract and concrete) and a strict type/instance view of how software constructs could be abstracted. This is reflected in the strictly linear approach to modeling that is still supported by the majority of modeling tools (in both branches of modeling).

Fundamental Weakness 5 (Two Level Modeling)**:** Today's tools are not only typically built on a linear modeling architecture where linguistic and ontological classification is tangled up, they also usually only make two levels accessible to end users – one level containing types and one level containing instances This forces modelers to use unnatural workarounds or ad hoc modeling features to represent subjects of interest containing deep characterization scenarios – that is, classification scenarios involving multiple levels of classification (which is the rule rather than the exception). The end result, as before, is the raising of the artificial problems encountered

in creating IT solutions, and an increase in the accidental[1] complexity embedded within visual models. This problem is common to both modeling traditions although it was first identified and comprehensively addressed in the constructive modeling tradition.

1.2 Research Goals

Users of visual modeling technologies stand to benefit enormously, therefore, from the introduction of a new, unified modeling framework which:

(a) breaks down the artificial barriers between these modeling traditions and makes their combined capabilities accessible in a seamless way with minimal unnecessary impedance and,

(b) provides fundamental and natural support for deep classification without the need for artificial workarounds and modeling constructs.

This is the essential goal of this thesis – **to demonstrate the feasibility of such a framework and develop a prototype environment supporting its use.** More specially, the concrete, high level goals of the research reported in this thesis were to develop a foundation for a unified modeling framework which supports -

Research Goal 1 existing SE modeling use cases in a way that makes it easier to create models with reduced accidental complexity compared to todays modeling frameworks,

Research Goal 2 existing KE modeling use cases in a way that makes it easier to create models with reduced accidental complexity compared to todays leading frameworks,

Research Goal 3 the co-use of SE and KE modeling use cases in as natural and simple a way as possible, making the uses cases traditionally available

[1] Accidental complexity is a term coined by Fred Brooks to characterize the complexity in a solution that arises from artificial shortcomings and weaknesses in the chosen implementation technology rather than from the inherent complexity in the problem.(22)

only to one of the communities available to the other as well with little if any impedance or accidental complexity,

Research Goal 4 deep characterization in as clean and natural a way as possible, facilitating the creation of models of deep classification scenarios with less accidental complexity than todays leading modeling tools.

1.3 Potency based Multi-Level Modeling

The shortcomings of traditional model frameworks for representing deep classification scenarios have been discussed for some time in the SE modeling community although they have only recently been recognized in the KE modeling community. There have been numerous approaches aimed at the unification of the communities (4, 39, 58) or bringing the benefits of one to the other (19, 35, 41). The pros and cons of the various possible solutions is the subject of an ongoing lively debate between modeling methodologists, and various concrete proposals have been put forward, ranging from the addition of ad hoc modeling features such as stereotypes and powertypes (36) to more radical restructuring of the underlying modeling infrastructure. One of the most well known is the potency-based, multi-level modeling approach proposed by Atkinson & Kühne in a series of papers. This proposes a fundamentally different architecture for modeling frameworks in which model elements are classified within two distinct dimensions – one linguistic dimension and one ontological dimension. Explicitly separating these two forms of classification allows tool-oriented classification to be separated from domain-oriented classification, so that the latter can include as many classification levels as necessary to best model the subject in hand. The architecture is therefore known as the Orthographic Classification Architecture (OCA), and employs the notion of potency to capture the ontological "typeness" of model elements (i.e. the degree to which they have the properties of types or instances or both).

The underlying premise for the research reported in this thesis is that the potency-based multi-level modeling approach supported by the OCA(10, 12) is not only capable of supporting the goals outlined in the previous section, it is the best approach for doing so. Various other enhancements of the original Potency based Multi-Level Modeling (PMLM) approach of Atkinson & Kühne(5) have been developed in recent years. Asikainen and Männistö (3) have proposed a formal foundation for the approach. Others, like Volz and Jablonski (69) or Varró and Pataricza (66) have developed practical tools to support it. And some like, Gitzel et al. (34) or Aschauer et al. (2), have adapted it for particular applications domains. However, they have all been developed exclusively to support the constructive, SE mode of modeling, and none of them are compatible or interoperable with another. The work described in this thesis is the first attempt to enhance and consolidate the PMLM approach of Atkinson & Kühne to support all the high level goals listed previously, especially the goal of unifying the SE and KE traditions of modeling.

1.4 Research Hypothesis

To achieve the stated goals, the research reported in this thesis explores the following hypotheses.

Hypothesis 1 It is feasible to enhance the original potency based multi-level modeling approach of Atkinson & Kühne to support high-level goals 1, 2, 3 and 4

To explore this hypothesis we developed a new, concrete foundation for PMLM, enhanced to support the use cases of the two main modeling traditions identified above. To support the visualization and enhancability use cases of the constructive modeling tradition the developed foundation is based on established metamodel technology (Eclipse Modeling Framework (EMF)(23)), but adapted to support the OCA of Atkinson & Kühne. To

support the reasoning and formal analysis uses case of the exploratory modeling tradition this foundation has been provided will full, formal semantics based on first-order predicate logic. This formalism sacrifices the guarantee of efficient decision problems provided by description logic for a wider range of possible reasoning services, even if some of these may sometimes not be efficiently computable. This new platform is accompanied by new notions of ontology well-formedness, validity and completeness in the context of multi-level modeling. The realized enhancements to PMLM are explained and motivated in chapter 2, while the underlying metamodel for the new platform, known as the Pan Level Model (PLM) is described in chapter 4. The accompanying formal semantics are described in chapter 6 and the enhanced notion of ontology well-formedness and correctness are elaborated in chapter 7.

Hypothesis 2 A powerful, general purpose modeling tool based on the PMLM enhancement referred to in Hypothesis 1 can support additional services and uses cases beyond those supported in the two main existing modeling traditions.

To explore this hypothesis a new, prototype modeling tool was developed, based on the Eclipse environment, that makes the capabilities of the platform developed to explore hypothesis 1 available to end users in a usable and accessible way. This included:

(a) the definition of a new concrete syntax for the enhanced approach to PMLM that allows multiple classification levels to be described in a seamless and uniform way, in accordance with established visualization metaphors,

(b) the provision of a large number of modeling services which support the use cases of the two main modeling traditions.

1.5 Structure

Chapter 2 provides an overview of the introduction to potency-based multi-level modeling (PMLM) and how it overcomes some of the shortcomings identified in the introduction.

Chapter 3 establishes the modeling use cases of both software and knowledge engineering and what requirements they impose on a foundation aiming to support them both seamlessly.

Chapter 4 contains the detailed description of the PLM, the linguistic metamodel developed in this thesis.

Chapter 5 introduces the level agnostic modeling language (Level Agnostic Modeling Language (LML)(16)). The LML is the concrete syntax developed for the PLM.

Chapter 6 defines the most fundamental building block of all the logic operations, namely the relation of instance to type, purely based on primary properties from the domain.

Chapter 7 and 8 use the results of the previous chapters to define powerful and novel end user services with a formal foundation.

Chapter 9 rebuilds and enhances two well known case studies from the SE and KE community to illustrate the usage and benefits of the results.

Chapter 10 investigates other approaches building on the same theoretical foundation and compares them against the research goals of this thesis.

Chapter 11 concludes by revisiting the research goals and hypotheses along with possibilities for future research.

Chapter 2

Multi-Level Modeling

In the context of software engineering, the term multi-level modeling covers any approach to modeling that aims to provide systematic support for representing multiple (i.e. more than two) ontological classification levels within a single body of model content. The motivation is to overcome fundamental problems in the traditional, linear, four-level modeling architecture popularised by the UML infrastructure and EMF, especially in the context of domains involving deep characterization (46). Some of these problems are immediately evident from the illustrative UML infrastructure diagram used in all recent version of the UML infrastructure specification to convey the architecture of the UML modeling framework (see figure 2.1).

2.1 Three Problems

One key problem is the so called **dual classification** problem which is highlighted by the model element `:Video` with value "2001:A Space Odyssey" for its attribute `title`, in the M1 level of the figure. In the domain of interest (i.e. the subject of the model) this entity is conceptually an instance of the concept `Video` with attribute `title` of type String, also at the M1 level in the diagram. However, in the language used to represent the model, the UML, the model element `:Video` is an instance of the linguistic (i.e. abstract

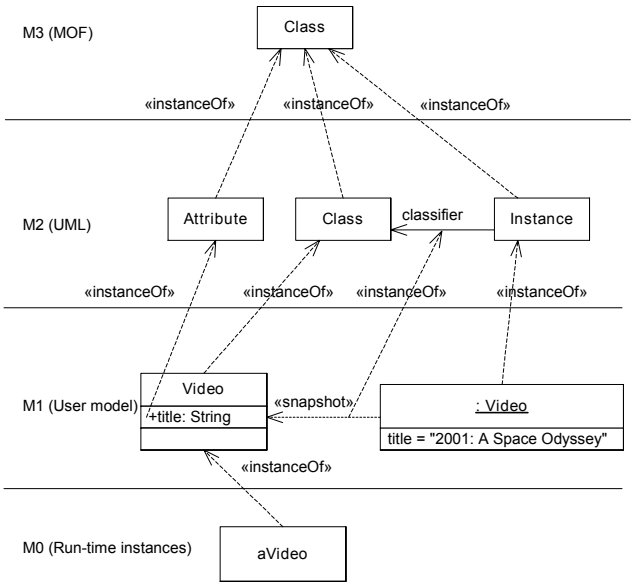


Figure 2.1: An example of the OMG four-layer metamodel hierarchy, taken from (51)

syntax) instance represented as a model element at the M2 level. Thus, `:Video`, like most model elements in general, is actually an instance of two types, one "linguistic type" capturing what kind of model element it is, and one "ontological" type representing the domain concept that characterizes its properties. However, the UML language and framework is only geared up to recognize one form of classification, linguistic classification. Ontological classification consequently has to be "programmed up" using various forms of workarounds such as stereotyped dependencies or associations (in the type instance pattern). Thus, in figure 2.1 `:Video`'s relationship to `Video` is modelled as a stereotyped dependency called "snapshot" even though it conforms to all the basic rules of classification in object-oriented modeling.

The second key problem is the so called **replication of concepts** problem which again is highlighted by the class `:Video` and `Video` at the M1 level. A fundamental goal of the UML infrastructure is to make all model-

ing content representable in, and interchangeable through, the Meta Object Facility (MOF) and derived serialization standards such as XMI(57). In other words, all model elements stored within the UML infrastructure are supposed to be classified by the types in the MOF so that they can be stored and interchanged in a uniform way. However, `:Video` and `Video` have no connection to the MOF model elements and thus cannot be regarded as being instances of anything in the MOF. Figure 2.1, therefore cannot be a valid classification based characterization of the UML, despite claiming to be so.

The third key problem is the **class/object duality** problem. The UML metamodel does in fact distinguish between model elements representing types (e.g. class) and model elements representing instances (e.g. instances) and therefore goes some way towards supporting the modeling of ontological classification relationships within the UML, but it only does so for one pair of ontological levels. However, many domains contain deep classification scenarios that involve more than two ontological classification levels. Again, the UML has no natural mechanism for handling this situation. Various artificial modeling constructs have to be used such as stereotypes and powertypes(52). These all provide some kind of backdoor mechanism for capturing metaclasses, but in a completely different way from the normal two level classification mechanisms.

The combined result of these problems is that UML models are usually more complex than they need to be because of the additional unnecessary modeling concepts that have to be learned and the awkward workarounds that have to be used.

2.2 Strict Metamodeling

The underlying reason for the problems in the UML infrastructure outlined above is the desire of the UML developers to adhere to a strict metamod-

eling architecture when defining the structure of the UML infrastructure. Strict metamodeling is the doctrine that results from the use of the classification relationship to define the levels within a model containing multiple classification levels.

Definition 1 (Strict Metamodeling)**:** All the types of a model element reside exactly one level above that model element. Since classification relationships connect instances to types they have to cross level boundaries. Moreover, they are the only kinds of relationships crossing level boundaries. All other kinds of relationship stay within one level.

Although conforming to this doctrine is not always easy (as the UML infrastructure models shows) doing so brings many benefits. A good analogy is the discipline of strong typing in programming languages which requires extra upfront effort to create programs adhering to the typing rules, but significantly reduces the average overall time involved in creating correct programs (when the extra debugging and testing time needed to reach the same level of quality with non-typed programs is taken into account). The alternative to strict metamodeling is "loose" metamodeling when all kinds of relationships can be drawn freestyle between all kinds of model elements in the style of RDFS and OWL Full models from the knowledge engineering community. Without the discipline of strict metamodeling it is much easier to create models which have incoherent structure and make many logically inconsistent statements. Moreover, the typical visualization of such models in tools such as Protege resemble spaghetti, with an overwhelming number of connections and fine grained model artefacts. This greatly increases the effort involved in understanding models and the chances for misunderstandings. The problems encountered by the UML infrastructure and other modeling frameworks such as the EMF when attempting to apply the doctrine of strict metamodeling stem from their failure to fully recognize and accommodate the two fundamental forms of classification when applying the doctrine.

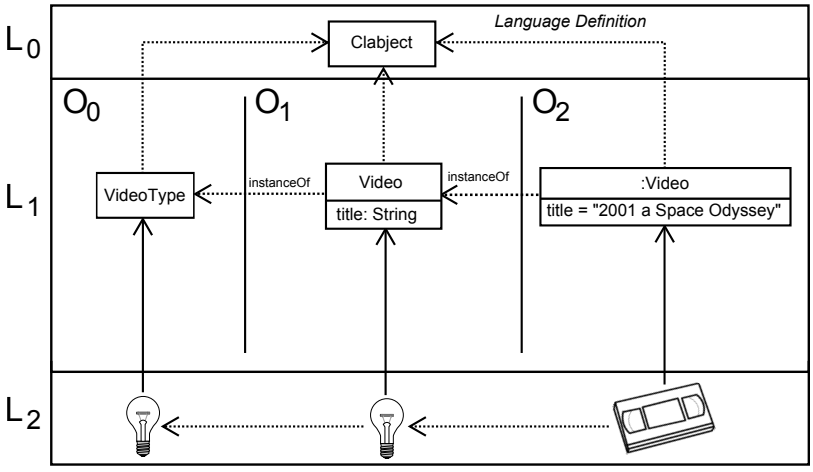


Figure 2.2: The orthogonal classification architecture

2.3 Orthogonal Classification Architecture

The orthogonal classification architecture (10) represents one of the most well know attempt to overcome the aforementioned problems whilst staying faithful to the basic principles of strict metamodel. It does this by basically separating the two forms of classification and organizing them in two independent dimensions, each adhering to the strict modeling tenets. In other words, it supports strict metamodeling, but in two "orthogonal" dimensions – hence the name.

As illustrated in figure 2.2, where the vertically arranged levels, L0, L1 and L2 represent the linguistic levels, and the horizontally arranged levels (within L1) represent the ontological levels, disentangling ontological and linguistic classification in this way provide the basis for overcoming all the fundamental problems identified above. From the point of view of `Video` and `:Video`, the organization of the model content is much the same as in the UML architecture shown in figure 2.1, except that the relationship between them is given proper recognition as a fully fledged classification relationship. The bottom level also plays the role of the "real world" whose

concepts are the subject of the domain model content in level L1. The two main differences between figure 2.2 and 2.1 are:

1. There is only one linguistic (metamodel) and it spans all ontological model content at L1 in a uniform and natural way. This solves the dual classification problem at a single stroke and gives the single underlying metamodel the correct relationship to all the model content which it is meant to store.

2. There are three ontological classification levels, all represented and related in the same, consistent uniform way. Moreover, the number of levels is not fixed at three. The PLM is completely agnostic to the number of levels and users can create as many model levels as they need to best describe the domain in hand. This removes the need for ad hoc modeling concepts like stereotypes or artificial workarounds using associations and allows deep classification scenarios to be modelled in the most natural way with minimum accidental complexity.

Disentangling the two forms of classification into two orthogonal dimensions is one of the key steps towards multi-level modeling. However, there are two other important concepts – one is the notion of clabjects and the other is the notion of potency.

2.4 Clabjects

The key to allowing modelers to include an arbitrary number of ontological levels in their models is to represent model elements (i.e. define linguistic classifiers) that are levels agnostic. The generally accepted way of achieving this is through the notion of clabjects (5). A clabject is the linguistic concept used to represent domain concepts (i.e. artefacts and connections) from the domain of interest.

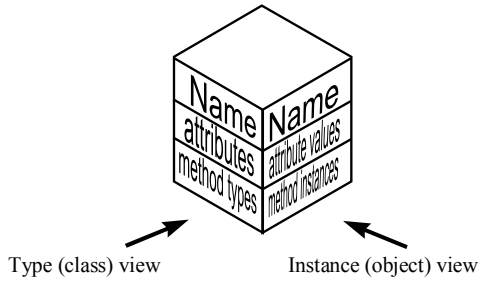


Figure 2.3: Cube Visualization of the two facades, taken from (6)

The key things that distinguish clabjects from traditional model elements supported in languages like the UML or OWL is that they are both types and instances at the same time. In other words, in general they have both a type facet and an instance facet. For example, `Video`, has both a type and an instance facet since it is an instance of `VideoType` on the left hand side and a type for "2001 a Space Odyssey" on the right hand side. Clabjects can also have attributes (previously called fields (15, 37)) and participate in connections. This is an unavoidable consequence of allowing model elements to be simultaneously instances of model elements at the level above and types of model elements at the level below.

2.5 Potency-Based Deep Instantiation

The downside of having type/instance agnostic linguistic classifiers like clabject is that some other mechanism is needed to capture the "typeness" of a clabject – that is, the degree to which it is capable of having instances. To achieve this Atkinson & Kühne introduced the notion of potency to indicate how many levels a clabject can have instances over (9). This original notion of potency is based on the following fundamental concepts -

1. potency is a non-negative integer associated with clabjects and attributes,

2. a clabject, i, instantiated from another clabject, t, has a potency one less than t,

3. the potency of an attribute of a clabject must be no greater than that of the clabject,

4. every attribute of a clabject, i, instantiated from a clabject t, must have a corresponding attribute in, t, with a potency one higher,

5. a clabject, i, instantiated from another clabject, t, must have an attribute corresponding to every attribute of t, except those that have potency zero.

"Corresponding" here means an attribute with the same name and datatype. Since potency values are defined as non-negative integers, it follows that a clabject of potency zero cannot be instantiated (i.e. represents an object in classic UML terms) and a clabject of potency one represents a regular class in classic UML terms. By allowing clabjects to have potencies of two and higher, new kinds of classification behaviour (spanning more than immediately adjacent levels) can be supported. This has been referred to as deep instantiation in (9). This interpretation of potency also lends itself to a simple definition of abstract and concrete classes. Since abstract classes cannot be instantiated they have a potency of zero, while concrete classes which can be instantiated have a potency greater than zero.

2.6 Multi-Level Modeling in Software Engineering

The four key principle outlined above:

1. strict metamodeling,

2. orthogonal classification architecture,

3. clabjets and

4. potency-based deep instantiation

were developed by Atkinson & Kühne in a series of papers from 1997 to 2003 (5, 6, 8, 9, 10, 11, 12, 14). We refer to this approach as "classic multi-level modeling". Other related ideas include powertypes (36) or materialization (60). Although the authors outlined the basic principles of PLM, they did not provide:

1. a formal semantics,
2. a comprehensive linguistic metamodel,
3. a comprehensive concrete syntax and
4. a prototype tool or platform.

This thesis extends classis Multi-Level modeling (MLM) in all four areas. Chapters 6, 7 and 8 provide formal semantics not only for the metamodel used, but also the end user services the prototype implements. The metamodel is introduced in chapter 4 and the concrete syntax is the subject of chapter 5.

Various other authors have also extended classic MLM in various ways to address some of these deficiencies. For example, Asikainen and Männistö (3) equip their metamodel with a mapping to an efficient computation language, while Lara (47) implements a textual concrete syntax and provide at least a command line tool. Volz and Jablonski (69) have implemented a visual editor for multi-level modeling. These are discussed further in chapter 10.

2.7 Multi-Level Modeling in Knowledge Engineering

As already mentioned above, KE languages already support MLM in one sense, but in a loose non-strict way. For example, Figure 2.4 is a representation of the `Video` example in RDF native syntax and the bubble notation

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
   <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
   <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
   <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
   <!ENTITY Ontology1327685302042 "http://www.semanticweb.org/ontologies/2012/0/Ontology1327685302204.owl#" >
   <!ENTITY Ontology1327685302204 "http://www.semanticweb.org/ontologies/2012/0/Ontology1327685302204.owl#2001" >
]>
<rdf:RDF xmlns="http://www.semanticweb.org/ontologies/2012/0/Ontology1327685302204.owl#"
    xml:base="http://www.semanticweb.org/ontologies/2012/0/Ontology1327685302204.owl"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:Ontology1327685302042="http://www.semanticweb.org/ontologies/2012/0/Ontology1327685302204.owl#"
    xmlns:Ontology1327685302204="&Ontology1327685302042;2001"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <owl:Ontology rdf:about="http://www.semanticweb.org/ontologies/2012/0/Ontology1327685302204.owl"/>
    <owl:ObjectProperty rdf:about="&Ontology1327685302042;instanceOf">
       <rdfs:domain rdf:resource="&Ontology1327685302042;Clabject"/>
       <rdfs:range rdf:resource="&Ontology1327685302042;Clabject"/>
    </owl:ObjectProperty>
    <owl:DatatypeProperty rdf:about="&Ontology1327685302042;title">
       <rdfs:domain rdf:resource="&Ontology1327685302042;Video"/>
       <rdfs:range rdf:resource="&xsd;string"/>
    </owl:DatatypeProperty>
    <owl:Class rdf:about="&Ontology1327685302042;Clabject"/>
    <owl:Class rdf:about="&Ontology1327685302042;Video">
       <rdfs:subClassOf rdf:resource="&Ontology1327685302042;Clabject"/>
    </owl:Class>
    <owl:Class rdf:about="&Ontology1327685302042;VideoType">
       <rdfs:subClassOf rdf:resource="&Ontology1327685302042;Clabject"/>
    </owl:Class>
    <owl:NamedIndividual rdf:about="&Ontology1327685302042;2001aSpaceOdyssey">
       <rdf:type rdf:resource="&Ontology1327685302042;Video"/>
    </owl:NamedIndividual>
</rdf:RDF>
```

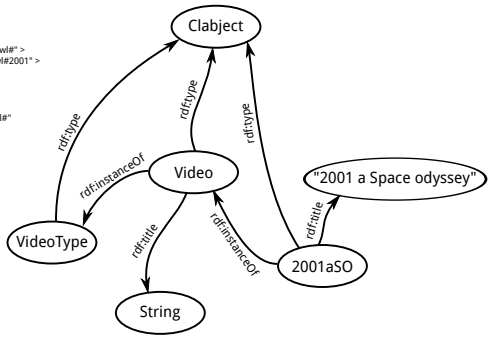


Figure 2.4: A representation of data in rdf and visual syntax

often used for graphical representation. From these diagrams the advantages of the more disciplined, strict visualizations of MLM should be clear. OWL Full also allows classification relationships between classes, but again in an unstructured, non-strict way. OWL DL, which is the flagship language for KE, and the basis for its attractive reasoning services, is essentially based on a two-level linear architecture. However, recently members of the KE community have recognized the potential value of MLM in KE(41).

2.8 Terminology

Enhancing classic MLM not only involves the consolidation of existing concepts and the addition of new ideas it also involves the refinement of terminology. In this section we introduce new terminology for concepts that exist in classic MLM. The terminology for brand new concepts is introduced along with the ideas when they are first explained.

Level → Model Previously, the whole collection of elements was called a model. In the course of time it became clear that the ontological levels provide the natural bounds to what is classically referred to as a model. So the set of all the elements residing at one ontological level is called

the model of that level. This conforms to common UML terminology since UML class diagrams, which are typically referred to as models, usually contain just types at one classification level.

Model → Ontology In the classical sense, an ontology embraces individuals, their types, all the properties and datatypes. With multi-level modeling in the picture, the equivalent is the stack of all levels (now called models). So an ontology is the union of connected models.

Field → Attribute In Classic MLM publications, the ontological attributes of clabjects were called fields. However, as the technology was refined for the wider modeling community it became clear that this name did not fit well with established terminology. The term attribute is therefore once again now used to refer to ontological attributes of model elements, but with a corresponding name change to linguistic attributes to avoid any confusion. An attribute is thus a triple of (name, datatype, value) belonging to a clabject.

Linguistic Attribute → Trait In classic MLM there was no special name for attributes of the linguistic metamodel types, so they were usually just referred to as linguistic attributes. To reduce the potential for confusion with ontological attributes we now refer to linguistic attributes as traits.

Feature Potency → Durability Initially, the trait that indicated how long a feature lasts in an instantiation tree was also called potency as with clabjects, because it basically means the same thing. When formalizing the meaning of classification, however, it became clear that potency plays a major role in classification relationships, but only for clabjects, not features. The clabject potency is far more important than the feature potency, and so there was a need to reflect this fact in the terminology to reduce the potential for confusion.

Value Potency → Mutability As with feature potency, the lifespan of a value is also less important than the potency of a clabject and should be named differently. While durability defines how long a feature "endures" in the classification tree, mutability defines how often its value can be changed over this tree.

Level numbering The original level numbering scheme defined by the Object Management Group (OMG) starts at the most abstract level with number three and advances downwards towards the most concrete level with number zero. Unless one allows negative numbers as level labels, however, this inherently limits the number of levels that can exist. Since one of the goals of this work it to remove this constraint, the traditional numbering scheme is not appropriate. In this thesis, therefore, the opposite numbering scheme is adopted with the most abstract level being labelled zero and the subsequent less abstract levels having successively higher integer label. The terms "higher" and "lower" are still used to refer to abstraction level rather than integer labels.

Chapter 3

Modeling Use Cases

For many years there was little interest in the similarities and differences between the various structural modelling paradigms, but with the recent confluence of end-user software applications and the Internet, typically driven by databases, this situation has changed. Understanding the accidental and essential difference between these paradigms has now assumed great importance. However, most previous comparison have focused on the idiosyncrasies of the languages typically used in each community, such as UML, OWL or ER diagrams. This chapter investigates the difference from the perspective of the use-cases of end users, and characterizes these differences in terms of the distinct modes of modeling. To provide the foundation and motivation for the concepts presented in the following chapters, in this chapter we investigate the different use cases predominant in the software engineering and knowledge engineering communities and explore how they can understood within a single unifying conceptual framework.

3.1 Traditional Modeling Use Cases

Although the software engineering and knowledge engineering (i.e. artificial intelligence) communities emerged independently with different priorities and foci, there has already been a great deal of cross fertilization between

the two communities, and there are now numerous tools and approaches for bridging the two technologies and translating between the two types of models (42, 59, 63). The main difference between them is the chronological order, or direction, in which the user populates and uses the contents of a model. In software engineering, the aim is to define new types and create instances which are constructed from those types. In knowledge engineering, on the other hand, one of the major goals is to establish new correlations between existing types by investigating individuals. Thus, software engineering uses case for models can thus be characterized as *constructive* whereas knowledge engineering uses case can be described as *exploratory*.

3.1.1 Constructive Modeling in Software Engineering

In a traditional software engineering oriented modeling tool, the user starts with a blank model and a palette showing the linguistic elements (in most cases UML) he/she can use to populate the model. The created diagram is called a class diagram. Every time a new element is created, the user has some options to configure the linguistic attributes (traits) using a context menu or a side pane of the main window. The elements are rendered in the concrete syntax of the used linguistic metamodel (again, UML).

Once a model (i.e. class diagram) is complete, the user can either let the tool create program code from the model or instantiate some types in the tool. The tool usually generates classes in an object-oriented programming language, from which the user can then implement some of the method stubs, adjust the code to the target platform or use the classes to create instances of the types in a running system. In an object diagram, the user can create instances of the previously defined types. The resulting instances will have exactly the properties defined by the types. If the tool does not automate the process of instantiating types the user must ensure that any instances have the properties specified. The fact that the instances have that type is a primary fact that is not challenged. There is no way to instantiate instances

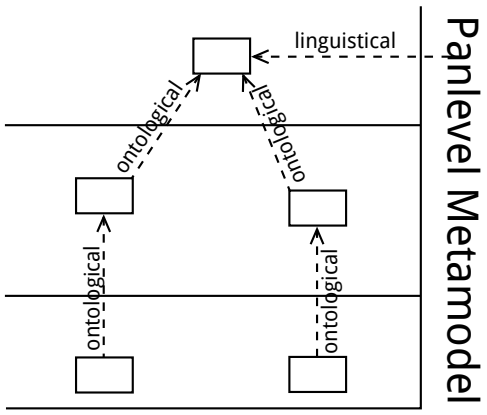


Figure 3.1: Typical Constructive way of modeling

further, so the classification is limited to one instantiation step. Since tools are used for visualizing the concepts rather than validating them, there are usually no reasoning services present. Object Constraint Language (OCL) constraints may be added to the types, but they cannot be validated against instances as most of the time there will be no instances present. They will however be used for code generation. There are mappings from OCL to the target programming language, so the constraints will result in executable code against the running instances. So the constraints are more than pure text labels — they are used to support software construction services such as code completion.

In constructive multi-level modeling, all elements on consecutive levels are created by ontological classification of the previously created elements. As every element is directly constructed from one special type, "its type", there is no need for checking or discovering classification relationships. If the instantiation mechanism works correctly, the created classification relationships will all be valid by construction.

The creation of the model is an ongoing process. So at any given point in time, the information contained in the model may not be complete and therefore any assessments based on the current state of information will

be limited. If the user enters a potency value to define the lifetime of the element, its instances are not yet present. A tool displaying a tooltip at that point telling the user that the potency value does not match the information in the model is basically useless if not annoying. So in constructive modeling, the tool can only validate present information against contradictions with other present information. Figure 3.1 shows a schematic view of a typical constructive modeling process.

3.1.2 Exploratory Modeling in Knowledge Engineering

The landscape of traditional knowledge engineering tools is much more diverse than for software engineering. There is also no standard language for representing knowledge models. OWL is one of the predominant ones, but there are other mainly text based representation formats as well. What all the representations have in common is their lack of sophisticated visual representation of model elements. Because they often have to deal with large quantities of data, they have a very efficient textual syntax, but no concrete graphical syntax.

The initial step is the creation of (part of) a domain model and the ongoing process is the discovery of correlations between the elements. So the average use case of modeling is not creating a model from a clean sheet of paper, but to work on an existing set of data to gain new insight into the domain. The data can be read in from a variety of data sources: data feeds crawling the semantic web, database records or other kinds of output data produced by a running system. New elements are created by instantiating linguistic metamodel elements at the appropriate ontological level. The elements are usually not instantiated from an ontological type and therefore do not have one distinct type. Also, if a classification relationship is discovered, it is common that the instance has more properties than those required by the type, as it was not created from that type. For the same reason, there is a need to check existing classification relationships. The connected elements

may not be natural instances in the sense that they possess all the properties required by the type, a case that does not make any sense in constructive modeling.

Knowledge engineering tools typically offer two main services to the user: subsumption and classification. Subsumption organizes the classes in a type hierarchy based on their formal description. A subtype is always a stricter description than the supertype. In other words, an instance of the subtype will always be an instance of the supertype, but not vice versa. The second operation can classify an individual as belonging to the extension of a type, based on its intention. The types are not modeled in the sense of software engineering tools, where attributes are added through a GUI, but in a more formal way by describing rules that an individual has to adhere to in order to be an instance of the set. The model elements in the model are assumed to be complete and the task of modeling is to enrich the model with correlations. More specifically, the information is already present in the model, but implicitly contained in the artefacts and not explicitly shown through correlations. The creation of these correlations is the primary purpose of exploratory modeling. A modeling tool can indicate invalid correlations to the user or offer services such as displaying the full inheritance hierarchy on a set of elements.

3.1.3 Bounded and Unbounded Modeling

In addtion to the "direction" in which models are populated, the advent of multilevel modeling gives rise to a second fundamental question related to the use case in which a model is applied: how many ontological levels are there?

In UML style modeling there has traditionaly only been one explicit level under development at a given time, and instanceOf relationships were modeled inside that level by the use of various ad-hoc mechanisms (e.g. stereotypes, powertypes etc.). The number of ontological levels was implied

by the use of those mechanisms. With the concepts of explicit ontological levels and potency to control a model element's influence on other levels there is the need to decide on the number of levels when designing the domain. A potency of two means that an element can be instantiated over two consecutive levels. The total number of levels is not constrained by the potency of the top level, as new elements (without a relation to any higher level) can be created with any potency.

In principle there is no fundamental reason to stop the user from creating further clabjects with higher potencies, therefore allowing more levels to come into existence. However, numeric potencies make a precise statement about the number of levels the element can influence. If the user does not want to make such a statement, there is the need for a special potency value, meaning "unbound", "unlimited" or "undefined". The concrete syntax for this value is "*" in line with the notation for unlimited multiplicity. If a clabject has * potency it means that the number of instantiation steps that can be performed from it is not limited. An instantiation of a * potency clabject can lead to a * potency clabject again or to a clabject with numeric potency. Once the potency turns numeric, there is no way of going back to *.

* potencies enable users to leave the scope of the whole model open to future extension, presenting an unbound dimension of modeling. A model with * potencies can be used to define a template or framework that can be instantiated across an unforeseen number of levels. Main applications include reuse of well-defined levels in various scenarios or the definition of a framework where the main purpose of the model is to be extended in any way by future users.

A model containing * potencies is called an unbounded model and a model without any * potencies is called bounded. If * potency is used, the scope of the stack of model levels changes. Fixed potencies produce a bounded model stack, whereas an undefined potency defines an unbounded

model stack. The dimension determining whether or not the model is bounded is called the *scope* dimension.

3.2 Modes of Modeling

Since the "scope" and the "direction" concerns discussed above are essentially independent there are basically four distinct combinations of fundamental use cases, giving rise to the two dimensional characterization of modeling modes illustrated in figure 3.2. The following subsections discuss the implications of these modes on the basic ingredients of multi-level modeling and explains how they need to be enhanced to support them.

3.2.1 Bounded Constructive Modeling

Bounded constructive modeling means there are only numeric potency values and that only elements at the top (most abstract) level have no ontological blueprint. All elements at all lower (less abstract) levels are instantiated from ontological types). As a consequence, the number of levels is fixed in the top level and the direction the model elements are populated is strictly top-down. The potency of a clabject therefore defines the potential depth of the instantiation tree. Upon instantiation, the potency of an instance is lowered by one with respect to its type. The only place were potency is defined by the user is the top level. All other potencies are correct by construction as potency assignment is performed automatically by the instantiation mechanism.

3.2.2 Unbounded Constructive Modeling

Unbounded constructive modeling means that some of the elements have * potencies, usually at the top level. As a result the number of levels is not fixed from the beginning, but the creation direction is still top-down. Potency also still defines the potential depth of the instantiation tree, just

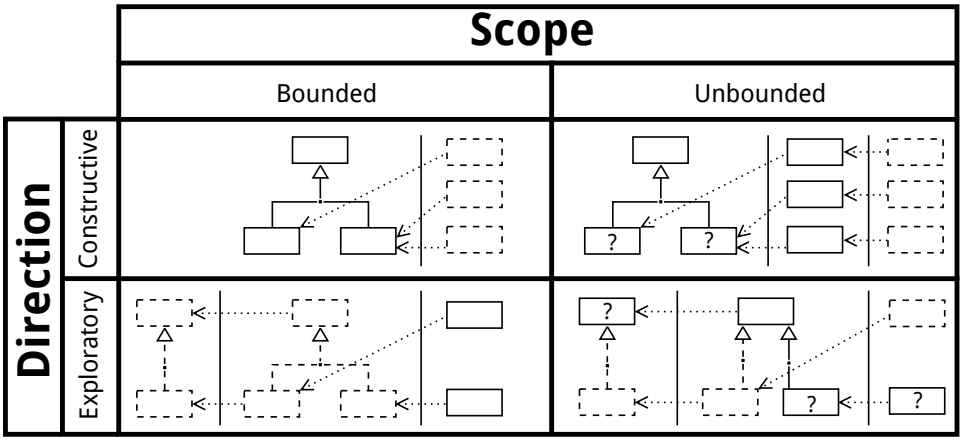


Figure 3.2: Scope and Direction of modeling

as with bounded constructive mode. The difference is that for * potencies of the top level, the default is to spawn * potency instances, the user then has the choice to change these potencies to any numeric value. Once the potency has a numeric value the normal rule apply for any further instantiations.

3.2.3 Bounded Exploratory Modeling

Bounded exploratory modeling has only numeric potencies but the direction of model creation is arbitrary. Types can be constructed with or without ontological types at any level. The total number of levels however is fixed in the sense that for every clabject its influence down the instantiation tree is limited. Since most elements are created without an explicit ontological type, there is no guarantee that type information is available for each clabject. In fact, the record of which clabject another clabject was instantiated from is historical information that can not be deducted from the properties of the model elements alone. If an instance possesses exactly the properties of two types (which need not to be equal) it cannot be judged whether or which one of the two is the type the clabject was instantiated from. Since the information in an exploratory model is assumed to be descriptive rather

than prescriptive, potency specifies the **actual** and not the **potential** depth of the instantiation tree. Together with the fact that instantiation information is historical, the instantiation tree is redefined to be the tree of instances that have exactly the properties required by the type.

3.2.4 Unbounded Exploratory Modeling

Unbounded exploratory modeling behaves exactly like bounded exploratory modeling except that the last constraint on the lifespan of clabjects on the instantiation tree is removed. Unbounded exploratory modeling is the most general modeling mode and has all the features available to the user. As it is the most general, it is the hardest to define precise use cases for. The meaning of potency is the same as in bounded exploratory mod but with the additional possibilities that * brings into the picture. Potency still defines the **actual** depth of the tree of possible instantiations, but the * value overrides the constraint in the sense that it represents any number and therefore is always valid. In a sense, therefore, it captures the notion that the exact number of levels is unknown.

3.3 Multi-mode Modeling

Traditionally, the different modeling communities have developed models using only one of the modeling modes described above. However, this was due to the limited awareness of the full range of modeling possibilities. The mode is not defined by the use case, but by the assumptions and actions performed. As a consequence the mode may switch multiple times throughout the model life-cycle, often rapidly in short periods of time. The mode switches are also implicitly triggered by the user and most of the time not even noticed.

A modeling tool supporting these modes therefore has to be able to switch between the modes very quickly and without direct user request. For

example:

1. entering * potency on the top level implies unbounded mode,
2. removing the last * potency implies bounded mode,
3. changing the potency on an element not on the top level implies exploratory mode,
4. invoking the instantiate operation of an element implies constructive mode,
5. linguistic creation of classification relationships implies exploratory mode and
6. linguistic clabject creation not on the top level implies exploratory mode.

Ideally the tool would provide some visual feedback as to which mode the model is currently in, but the general operation is not constrained. In other word, the user is not limited in his actions, but his actions may trigger consequences to the modeling mode (and therefore to the semantics of potency) that he may not be aware of at first sight. If these semantics do not match the current user intention, the mode can be switched at any time, but of course an action can trigger a mode change away from the current mode. For example if the user wants potency to be the **potential** depth of the instantiation tree, i.e. a potency of 1 is not an error on a clabject without instances, he can set the mode to constructive. If the user then creates a clabject by linguistic instantiation on the leaf level, the model will switch back to exploratory and the potency one clabject will be an error. Of course, the tool can provide a parameter to lock the mode, with the implication that certain semantic operations are not guaranteed to work.

3.4 Important Dichotomies

Information in an ontology can be interpreted in different ways depending on the context and the uses cases which the modeler has in mind. For example, some model elements can be entered directly by the user, while others can be added by an inference engine. As another example, when analysing an ontology, and responding to queries, some information may be given greater importance than (i.e. may override) other information. Also of critical importance is the way in which absent information is interpreted. For example, absent infromation can be interpreted as false (closed world assumption) or it can be interpreted as unknown (open world assumption).

These concerns give rise to some important dichotomies that govern the way information in an ontology is interpreted:

Definition 2 (Primary versus Secondary Information)**:** Primary information Primary information is information whose validity cannot be questioned. If there is a conflict between two pieces of information in an ontology, one of which is primary and the other secondary, the secondary information is automatically assumed to be incorrect. For example, if a modeled classification conflicts with a disjoint generalization, and they have different weight, the primary information is taken to be correct (e.g. the generalization) and the secondary information incorrect (e.g.the classification).

Definition 3 (Expressed versus Computed Information)**:** Expressed information is information that has been input into the model by the user. Computed information, on the other hand, is information that has been added to the system by some kind of automated reasoning engine or transformation. In other words, it is information that has been automatically computed rather than directly expressed by the user. Expressed information is often primary, and computed information is often secondary, but this is not always the case.

Definition 4 (Open world versus Closed world)**:** Especially in the knowledge

engineering community, the distinction between an open or closed world assumption is very important. Open world means that if there is not sufficient information present to answer a question the answer is undefined. In contrast, closed world means that if there is not sufficient information to answer a question positively it answered negatively. Constructive modeling traditional takes place under the closed world assumption, whereas exploratory modeling typically takes place under an open world assumption.

Definition 5 (Ontological versus Linguistic Information)**:** Ontological pieces of information are statements about observable facts in the subject domain. In contrast, linguistic pieces of information are statements in the used modeling language which do add to the description of the subject domain. For example, setting the level or potency of a clabject is linguistic information whereas the creation of an entity is ontological information.

Chapter 4

The Pan Level Model

This chapter informally introduces the PLM in natural language. The pan level model is the linguistic metamodel at the level L0 of the OCA illustrated in figure 2.2 on page 41 and defines all the linguistic types used to represent ontologies at the L1 level. After a detailed description of the model elements and their relationships the operations for navigating and querying ontologies are defined.

4.1 Design Goals

The underlying goal of the PLM is to provide the backbone for a modeling framework that addresses the observed weaknesses (see section 1.1) of the UML and the UML modeling infrastructure outlined in chapter 2, whilst retaining their strengths and benefits. More specifically, the PLM was designed to:

Be level agnostic. Many of the problems in the UML originate from the different representation formats for types and their instances. Classes have a different representation than objects, as have Associations and links. For a connection between two types, there are three elements (one Association and two AssociationEnds) and on the instance level

only one Link. The PLM tries to overcome these difficulties by representing elements on different ontological levels in the same way.

Support the OCA. The Orthogonal classification Architecture forms the theoretical foundation of the PLM. The PLM follows the OCA by defining an orthogonal linguistic layer spanning all ontological levels. Deep classification is supported by element traitsfor all relevant elements.

Enable Reasoning. Knowledge Engineering is one of the uprising disciplines in modeling and adds a great deal of value to every model. The key to enabling reasoning and formal processing of any kind is a formal specification of all the elements involved. If the knowledge contained in a model can be expressed by the model framework using set theoretic constructs, the user gains access to a large number of mature computation services working on the knowledge. Therefore every PLM element has a complete and formal specification not only of its syntax, but also of its semantics.

Support Constructive and Exploratory Modeling. During the research that lead to the design of the present PLM it came to surface that the differences in the communities around modeling lie not only in the questions they want answered, but also in the way the models comes to life and evolves. Certain parts of the model have a different meaning when looking at them from a constructive or exploratory perspective. The PLM is not only aware of that, but embraces the different modes by explicitly supporting them and making their differences visible to the user.

Be as simple as possible. During the design of a language the designer has to take decisions affecting sometimes contradicting dimensions:

Easy of use, clean design, efficient implementation, minimal complexity. The PLM tries to have as few elements as possible, therefore choosing simplicity over efficient implementation.

Be UML like. Despite its weaknesses, the UML is the de facto standard for general purpose model representation, and with good cause. The UML way of rendering classes and concepts like generalization or multiplicity has been very successful. So in the concrete syntax the PLM tries to reuse as many of these positive concepts as possible. For the concepts that go beyond the original scope of the UML, like potency for example, PLM tries to weave the concrete syntax into the UML style as seamlessly as possible. The general guidelines for concrete syntax definition follow the UML best practices: straight lines with minimal junctions and drawable in black and white by humans on paper/whiteboard with one pen.

4.2 Types

In the following sections there are a lot of statements about types and instances. In the traditional UML style of modeling, the user would model the types, generate code from the diagram and have the instances created by the execution of a program. In a multilevel modeling environment, the concepts of type and instance are expanded as instances are part of the model and elements in the central ontological level are both instances and types.

4.2.1 Informal Definition

So what is a type? A type is a definition of how an instance has to look like. In other words it is a set of constraints another element has to satisfy to be an instance of the type. This definition is not given explicitly by stating it in a syntax like OCL(54, 73, 74), it is given implicitly by the properties the type defines. For every feature with potency zero a type possesses, the instance

has to possess a conforming feature. For every connection with potency greater than zero and that is mandatory (multiplicity greater than zero) that the type participates in, the instance has to participate in a connection which is an instance of the type's connection. The ontological properties form the major part of a types definition, but not the complete definition. The other part of a types definition are the values of its traits.

4.3 Metamodel

The following section contains a listing of all the elements of the PLM with an informal textual explanation of their syntax and semantics. Every generalization in the metamodel is disjoint and complete. So in an ontology there can never be an element which is not a linguistic instance of one of the concrete types and there can never be an element that is a linguistic instance of more than one concrete type.

4.3.1 Abstract Metamodel Elements

Abstract elements cannot be instantiated directly, only through subtypes.

4.3.1.1 Element

element is the top of the inheritance hierarchy. Every artefact and correlation inherits from element. The direct subtypes of element are ontology, model and ownedElement.

Traits

name : String[0..1] The general identifier of every element. As there may be anonymous elements, the name is optional.

expressed : Boolean Boolean switch indicating the origin of the element. An expressed element was created explicitly by the user. A not-expressed (a.k.a. computed) element was not been created by the user

but computed by some kind of inference process. Often the newly created element will make information explicit that was implicitly defined within the model before, but there may also be rule based transformations introducing new computed information.

4.3.1.2 OwnedElement

An ownedElement is an element that is owned by another element to provide the natural concept of containment. OwnedElement is a subtype of element and a supertype of artefact and correlation. It does not introduce any traits but is used to specify the children types for elements which can be an owner.

4.3.1.3 Artefact

An Artefact is an element that represents a concept/idea of the modeled domain. Artefact is a subtype of ownedElement and a supertype of clabject and feature.

4.3.1.4 Property

Property is the supertype of features and roles. As such it is the umbrella for type definitions and constraints. Every ontological statement a type makes is a property. Besides its traits, a type can only influence instances with properties. Property is not an element, but all its concrete subtypes are.

4.3.1.5 Feature

A feature extends a clabject with either data or behaviour. Feature is a subtype of artefact and property and a supertype of attribute and method. The key identifier of a feature inside a clabject is its name.

Traits

durability : Integer or * The durability (or feature potency) of a feature controls whether or not a feature is passed on to the instance when the containing clabject is instantiated. If the durability of a feature is greater than zero when the clabject is instantiated, the feature is passed on to the instance. If the durability equals zero, it is not passed on. The durability of a feature can never be negative. * is interpreted as greater than zero but undefined and bigger than any numeric value. If a feature with integer durability is passed to an instance, the resulting feature will have an integer durability that is exactly one lower than the originating feature. A feature resulting from a numeric durability can never have * durability. If a feature with * durability is passed to an instance, the resulting feature can also have * durability or any (non-negative) integer durability.

4.3.1.6 Clabject

Clabjects are the main building blocks of PLM models. Clabject is a subtype of artefact and a supertype of entity and connection. Clabjects have the ability to act both as a type for their ontological instances and as an instance of their ontological types. Clabjects are the only elements that can be instantiated or enter correlations. Each concrete clabject defines the set of its instances. Clabjects can contain other artefacts, so, as clabjects are artefacts themselves, clabjects can contain clabjects to model ontological composition. Clabjects also contain the features which provide them with data and behaviour. Navigations to other clabjects are captured via roles.

Traits

level : Integer The level of a clabject indicates the model (i.e. ontological level) the clabject resides in. The container around an ontological level is a model, but the actual level is defined by the contained clabjects.

potency : Integer or * The potency of a clabject has different meanings with regards to the modeling mode. In both cases the potency makes a statement about the depth of the isonym tree of the clabject.

exploratory modeling mode The potency states the *actual* depth of the isonym tree.

constructive modeling mode The potency states the *potential* depth of the isonym tree.

The * value is the defining characteristic of the unbounded modeling mode. When an instance is created from a clabject with numeric potency, the instance's potency is one lower than the potency of its type. If the type has potency *, the user may opt to give the instance potency * as well. A clabject with numeric potency cannot spawn instances of * potency. The depth of the *current* isonym tree is computable and makes up the only valid value for potency in exploratory mode. In constructive mode, the information in the model is not complete, so the potency value may not be reflected in the current depth of the isonym tree. This indicates that more instances are expected some time in the future.

children : Artefact[*] Clabjects are domain containers, i.e. they can contain other artefacts. The children of a clabject are the artefacts owned by the clabject.

4.3.1.7 Correlation

Correlations are the counterpart to artefacts. Correlation is a subtype of ownedElement and a supertype of generalization, classification and setRelationship.

4.3.1.8 SetRelationship

SetRelationships indicate correlations between two sets that are represented by clabjects. SetRelationship is a subtype of correlation and a supertype of inversion, equality and complement. Each setRelationship makes a statement about a clabject relative to the relationship's base.

Traits

base : Clabject The base clabject for the relation the setRelationship describes.

4.3.2 Toplevel Elements

Top level elements are administrative concepts above the actual content of the ontology. They provide the needed infrastructure to start and maintain what in the user experience is called an ontology.

4.3.2.1 Ontology

Ontologies are the most general type of elements. Most of the times, the user will be modeling only one ontology at a time. An ontology is the container for the ontological levels (models). Ontology is a subtype of element.

Traits

children : Model[*] The children of an ontology are the models that the ontology spans

4.3.2.2 Model

Models represent one ontological level. So the number of models present in an ontology is equivalent to the number of modeled levels. Model is a subtype of element and ToplevelRenderingContainer. Each model contains all the elements of its level, either directly as a child or recursively in the

case of features and possibly clabjects. All Correlations inside a model are direct children of the model.

Traits

children : OwnedElement[*] The children of a model are the elements of the ontological level directly owned by the model.

4.3.3 Concrete Artefacts

Concrete artefacts are the elements the user builds his domain model with. There are only five, but in combination with ontological classification they provide the full expressiveness needed to model the domain of interest.

4.3.3.1 Entity

Entities are the main elements of any PLM model. Entity is a subtype of clabject. Entity does not define any additional traits to those of clabject. Therefore its existence is justified by having a name for concrete clabjects that are not connections.

4.3.3.2 Connection

Connections are the counterpart to Entities. Connection is a subtype of clabject. Connections connect clabjects, but can utilize the full potential available to clabjects themselves. They can even participate in other connections. Connections are first-class citizens of a model and therefore can enter classification and generalization relationships, be instantiated and contain other elements, even clabjects again. Each connection connects a number k of clabjects. k is called the order of the connection. For $k = 2$ the connection is called binary ($k \geq 2$ for all connections).

Traits

transitive : Boolean[0..1] A transitive connection has the constraint that if a is connected to b and b is connected to c then a is also connected to c. In the PLM, the statement of transitivity applies to the instances of the connection. So if a connection is transitive, then for its instances the rule of the connection from a to c holds. It can be computed whether or not a claimed transitivity holds in the classified level, so for any classified level, transitivity is either given or not. If the information given in the type is primary, the existance of further instances can also be inferred.

reflexive : Boolean[0..1] If a connection is reflexive, not only is the source connected to the target, but every participating clabject is also connected to itself. If a connection is not reflexive then a clabject participating in the connection can not reach itself via that connection.

symmetric : Boolean[0..1] If a connection is symmetric, every navigation is possible in both ways. So for every navigation enabling a to navigate to b with the roleName rN, there exists another navigation connecting b to a with the same roleName rN. The second navigation cannot be defined within the same connection, as the roleName has to be unique within one connection and therefore these two roles cannot exist in the same connection. If a connection is asymmetric, there must not be such a navigation.

4.3.3.3 Role

Roles are the immediate and mandatory supplements of connections. A role defines the participation of a clabject in a connection. Furthermore, a role also defines a navigation possibility for the other participating clabject. The number of roles equals the number of clabjects participating in the connection (and therefore the order of the connection).

Traits

connection : Connection The connection defining this role. A role only comes into existence with its connection, so each role is contained within exactly one connection.

destination : Clabject The target of this role. The target is the clabject playing the role. The destination is reachable (provided the role is navigable) by all the other participating clabjects of the connection (i.e. by all the other destinations of the roles of the connection).

navigable : boolean Boolean switch indicating whether or not the role is navigable. If a role is not navigable the relation still persists, but is not usable from the source's side. However it still remains one of the source's properties.

roleName : String The roleName the destination can be reached by. RoleNames are key for navigation. So in order to navigate to a destination, any source needs to specify the roleName it wants to navigate by. RoleName identifies the destination in the context of the connection, which means that in the roles of this connection there can be no other destination with the same roleName.

lower : int[0..1] The lower multiplicity bound.

upper : int[0..1] The upper multiplicity bound. The bounds either both exist or none exists. As multiplicity is a statement about the classified model of the current role, it exists only at roles which are defined by a connection with potency greater than zero. Although defined for the destination, the multiplicity is rather a constraint on whether the sources can reach the destination, stating that each source has to be able to reach at least *lower* and at most *upper* destinations.

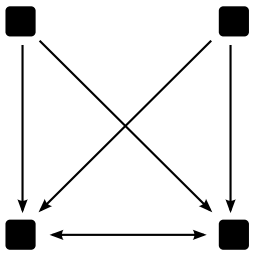


Figure 4.1: The navigations defined by one connection

When analysing a clabject, determining its roles is not trivial. A clabject does not store the connections it participates in. The role stores the information and, as one of the core PLM principles is not to have redundant information it is not stored again in the participating clabjects. To obtain all the navigation possibilities of a clabject all the roles inside the model have to be processed. Each of the connections which the clabject under analysis does participate in introduces navigations to those other participants.

From the perspective of the properties a type defines it is desirable to have a shorthand way of finding the navigations possible from a clabject. The roles of a clabject are made available through the operation *navigations()*. So a "navigation possibility", or just "navigation" for short, is the occurrence of a role in the result of the *navigations()* operation.

The number of navigations defined by one Connection Each connection defines a role for every clabject that takes part in it. Every participant that is navigable defines a navigation for all the other participants in the connection. Figure 4.1 shows a schematic example. The connection is not visible and the four dots represent the participating clabjects. The top two participants are not navigable. The bottom two participants form a mesh and the top two have navigations to every member of the mesh. So for one connection with four roles there are six navigations.

Let $\mathtt{s}$ be the order of the connection, $\mathtt{n}$ the number of roles which are not navigable, $\mathtt{n} < \mathtt{s}$ and $\mathtt{k}(\mathtt{s}, \mathtt{n})$ the number of navigations the connection

defines. For $\mathtt{n} = 0$ the connection defines a full mesh:

$$\mathtt{k}(\mathtt{s}, 0) := \mathtt{s}^2 - \mathtt{s}$$

For every connection, there are $\mathtt{s} - \mathtt{n}$ destinations which each have a navigation to every other destination, so in every connection, there are

$$(\mathtt{s} - \mathtt{n})^2 - \mathtt{s} + \mathtt{n} \tag{1}$$

navigations defined.

The non-navigable destinations do not define any navigations between each other, but each of those $\mathtt{n}$ destinations has a navigation for each of the $\mathtt{s} - \mathtt{n}$ navigable destinations. That is

$$\mathtt{n} * (\mathtt{s} - \mathtt{n}) \tag{2}$$

Navigations between non-navigable destinations and navigable destinations.

The addition of (1) and (2) defines the number $\mathtt{k}$ of navigations defined by any connection depending on its order and the number of non-navigable destinations.

$$\begin{aligned} \mathtt{k}(\mathtt{s}, \mathtt{n}) : &= (\mathtt{s} - \mathtt{n})^2 - \mathtt{s} + \mathtt{n} + \mathtt{n} * (\mathtt{s} - \mathtt{n}) \\ &= \mathtt{s}^2 - 2\mathtt{sn} + \mathtt{n}^2 + \mathtt{sn} - \mathtt{n}^2 - \mathtt{s} + \mathtt{n} \\ &= \mathtt{s}^2 - \mathtt{sn} - \mathtt{s} + \mathtt{n} \end{aligned}$$

Table 4.1 shows the number of navigations $\mathtt{k}$ for connections with order up to four.

4.3.3.4 Parameter

Parameters encapsulate artefacts or values participating in the body of a method. Inside the body, any parameter has a name that is local to the namespace of the method.

s	n	k(s,n)
2	0	2
2	1	1
3	0	6
3	1	4
3	2	2
4	0	12
4	1	9
4	2	6
4	3	3

Table 4.1: The number of navigations in relation to the size and number of non-navigable destinations.

Traits

name : String The name of the parameter in the namespace of the method body.

expression : Expression The expression evaluating to the value of the parameter at runtime. This value can be either a number, a boolean, any other primitive or complex value or a model element.

4.3.3.5 Method

Methods enhance clabjects with behavior. Method is a subtype of feature. The side effects and even the execution of a method are not defined in the PLM specification, but are a concern of the system that implements the PLM.

Traits

body : String The body of a method operates on a set of input parameters to produce a set of output parameters and possibly changes the state

of the clabject. Since is provides a textual description of the actions to be performed by the executing environment it has to be written in a language that the environment can process.

input : Parameter[*] A set of input parameters for the execution of the body.

output : Parameter[*] A set of output parameters produced or affected by the execution of the body on the input parameters.

4.3.3.6 Attribute

Attributes enhance clabjects with data. Attribute is a subtype of feature. Attributes connect a clabject to the value of a datatype via a name (the name of the attribute). Valid datatypes are primitives like Boolean, String, Integer, List or Set, but not other clabjects. Relationships to other clabjects should be modeled with connections.

Traits

datatype : Expression The datatype of the attribute. The expression evaluates to the datatype, which in most cases will be the name of the type. Other artefacts are not valid attribute types and have to be modelled with connections. The implementing environment may implement more sophisticated conformance rules to enable for example the conformance of BigInteger to Integer.

value : Expression The actual value of the attribute. The expression evaluates to the actual value. If values are compared and checked for equality, it is not the string value that is compared but the result of its evaluation. For example if one expression is “5” and another is “owner().features()->size()” those are equal if the owning clabject holds five features.

mutability : Integer or * Mutability is another form of potency, namely value potency. The mutability controls whether or not attributes created from the attribute may change their value. If the mutability is zero, they are not allowed to change their value but must retain the value given upon creation. In other words, the value is fixed at the type level and the same for all the created attributes. The mutability may never be higher than the attribute's durability. If an attribute is created, the mutability is lowered, but not below zero. It is possible (in fact the default use case for mutability) to create an attribute from a mutability zero attribute. A mutability can only be * if the attribute's durability is *.

4.3.4 Concrete Correlations

Concrete correlations are used to enhance the modeled clabjects with logic information. They always connect two or more clabjects in a logical way so that the affected clabjects can make use of the information for navigation on the one hand, and pose a general statement on the other hand.

4.3.4.1 Classification

Classification connects a type to an instance. Classification is a subtype of Correlation. Classifications are the only PLM elements that cross ontological level boundaries. Furthermore, every classification must connect two clabjects from adjacent ontological levels. For containment purposes a classification is regarding as residing on the level of the instance and is always contained by the model of the instance. The terms *isonym* and *hyponym* are used for the first time in the following paragraph. The formal definition is given in definition 30 on page 134. Informally, an instance of a type is an isonym if it does define the properties required by the type, and no more. An instance that does define more than the required properties, is called a hyponym.

Traits

type : Clabject The type of the classification.

instance : Clabject The instance of the classification.

kind : ClassificationKind ClassificationKind is an enumeration with four values

instance the instance is an instance of the type. What kind of instance is not known or not of relevance.

isonym the instance is an isonym of the type

hyponym the instance is a hyponym of the type

instantiation the instance is an isonym of the type and it was created by instantiation from the type. The type is thus the blueprint of the instance.

4.3.4.2 Generalization

Generalizations indicate subtype/supertype relationships between clabjects. Generalization is a subtype of Correlation. The cardinality of the traits supertype and subtype is dependent in the sense that only one can be greater than one.

Traits

supertype : Clabject[1..*] The supertypes (possibly multiple) of the generalization.

subtype : Clabject[1..*] The subtypes (possibly multiple) of the generalization.

disjoint : boolean[0..1] Indicates whether or not the generalization divides the subtypes into disjoint subsets. Disjoint can only be true if there is more than one subtype. The meaning is that any instance of the supertype can only be an instance of at most one of the subtypes. If the generalization does have more than one subtype and is not disjoint it means it is overlapping in the sense that there exists an instance of the supertype that is an instance of more than one subtype.

complete : boolean[0..1] Indicates whether or not the generalization covers all the instances of the supertype. In a complete generalization, any instance of the supertype must be an instance of at least one of the subtypes. In other words there can be no instance of the supertype that is not an instance of any of the subtypes.

intersection : boolean[0..1] Indicates whether or not the subtypes are an intersection of the supertype. If the generalization is an intersection, every clabject that is an instance of all of the subtypes is also an instance of the supertype. If the generalization is not an intersection, there exists a clabject that is an instance of all the supertypes, but not of the subtype.

4.3.4.3 Inversion, Complement and Equality

Inversion, complement and equality are the concrete subtypes of setRelationship. They each connect one clabject to another clabject and make some kind of statements about how the sets of instances they define are related to one another.

Inversion

inverse : Connection The inverse connection to a connection.

Inversion only makes sense for binary connections. An inversion states that the inverse connection inverses the original connection. In other words, for every instance of a connection connecting two clabjects from a to b there is an instance of another inverse connection connecting b to a.

Complement

complement : Clabject The complementary clabject to a clabject.

universe : Clabject The universe in which the one clabject complements another clabject.

Complement states that the sets defined by two clabjects are complementary in the set defined by universe. In other words, every instance of the universe clabject that is not an instance of one of the complementary clabjects is an instance of the other one. Complement statements are only valid inside the universe. So a complement has to be a subtype of the universe as a prerequisite. The original clabject does not need to be a subtype of the universe, however. If the original clabject is a subtype of the universe as well, then together with the complement it partitions the universe so an inference engine could discover a disjoint and complete generalization. Complement is not symmetric.

Equality

equal : Clabject A clabject equal to another clabject.

The meaning of equality is not well-defined in natural language but a precise definition needs a context to work in (e.g. "equal in functionality"). The context for the equality of PLM clabjects is the defined Properties. Two clabjects are equal if the Properties they define are equal, so the equality relation is symmetric.

Figure 4.2: The complete PLM metamodel with inheritance and traits rendered in UML.

4.4 Formalism

The previous sections provided an informal, metamodel-oriented description of the PLM in the style popularized by the UML. However, to provide a more rigorous definition of the semantics of the model elements and the operations that work on them, it is necessary to complement this with a more formal description. This section introduces the basic concepts that underpin the formal description of the PLM.

Element symbols Model elements are abbreviated by greek symbols. The linguistic set of all the elements of a type is shown in $\mathsf{sans-serif}$ font. The symbols used are shown in table 4.2.

first-order predicate logic The statements are given in first-order logic using the following symbols:

- quantifier symbols forall $\forall$ and exists $\exists$
- negation $\neg$. Some symbols have a separate negation syntax, e.g. $\neq, \nexists$
- logical connectives: conjunction $\wedge$ and disjunction $\vee$
- implication: $a \implies b$, meaning if a then b
- biconditional: $a \iff b$, meaning a if and only if b
- equality: $a = b$, meaning that a and b are the same
- colon as a selection: $a : a \neq b$ returns false for any a that is not b. The statement is usally used to select or filter elements $\forall a : a \neq b$ returns all the a's that are not b's.

set notation $\in, \notin, \cup, \setminus$ and $\cap$ are used as well as the construction of a set using curly brackets. $\{a \in A : a \neq b\}$ defines the set off all the elements in A that are not b.

pseudocode Where a single expression is not sufficient to express a meaning, the statements are chained in pseudocode. Control flow constructs used include for-all loops, if-then-else and return statements.

definition := is used to state that the left hand side is defined in terms of the right hand side.

element traits access If a trait of an element is accessed the trait is separated from the element by a dot and shown in $\mathsf{sans-serif}$ font. $a.\mathsf{b}$ accesses the trait `b` of the element a.

element operation access If an operation of an element is called, the operation name is separated from the element by a dot and the operation name is shown in `monospace` font. Even if the operation does not require parameters, brackets are shown after the operation name. $a.\mathsf{b}()$ calls the operation `b` on the element a.

Symbol	Meaning	Symbol	Meaning
χ	an Ontology	Σ_i	the model of level i
γ	a Clabject	Clabject	the set of all Clabjects
ϵ	an Entity	Entity	the set of all Entities
δ	a Connection	Connection	the set of all Connections
ψ	a Role	Role	the set of all Roles
η	a Feature	Feature	the set of all Features
ζ	an Attribute	Attribute	the set of all Attributes
π	a Method	Method	the set of all Methods
λ	a Correlation	Correlation	the set of all Correlations
ξ	a Generalization	Generalization	the set of all Generalizations
ϕ	a Classification	Classification	the set of all Classifications
υ	a SetRelationship	SetRelationship	the set of all SetRelationships
υ_i	an Inversion	Inversion	the set of all Inversions
υ_e	an Equality	Equality	the set of all Equalities
υ_c	a Complement	Complement	the set of all Complements
μ	a SetRelationship	SetRelationship	the set of all SetRelationships

Table 4.2: The symbols used for abbreviating model elements

4.5 Metamodel operations

The metamodel provides certain operations which do not introduce any new information into the metamodel, but simplify the use of the elements.

Definition 6 (Model ordering)**:** The order of the models in an ontology is given by the direction of the classification relationships. Given one model, the classifying and classified model is obtained by the adjacent levels:

Σ_i.`classifyingModel`() := Σ_{i-1}

Σ_i.`classifiedModel`() := Σ_{i+1}

Definition 7 (Border Models)**:** A model is the root model if there are no models that classify it.

Σ_i.`isRootModel`() := $\nexists\Sigma_{i-1}$

As a consequence, there cannot be classifications in a root model,

$\Sigma_i.\texttt{isRootModel}() \implies \Sigma_i.\mathsf{classifications} = \emptyset.$

Analogous to the root model, a leaf model cannot classify any other model.

$\Sigma_i.\texttt{isLeafModel}() := \nexists\Sigma_{i+1}$

As there are no types in a leaf model, generalization does not make sense. However, the existence of a generalization does not imply the existence of a classified model (as classifications do for root models).

Definition 8 (Model Inheritance)**:** With the help of generalizations, any clabject can return its supertypes. The generalizations to be processed are obtained from the model the clabject resides in (Operation 1). Analogous to

Operation 1 $\gamma.\texttt{modeledSupertypes}()$

```
result ← ∅
domain ← Sigma_{γ.level}.children ∩ Generalization
for ξ ∈ domain do
  if γ ∈ ξ.subtype then
    result ← result ∪ ξ.supertype
    for γ_s ∈ ξ.supertype do
      result ← result ∪ γ_s.modeledSupertypes()
return result
```

the supertypes, there can be an operation to retrieve the modeled suptypes of a clabject.

$\gamma.\texttt{modeledSubtypes}() := \gamma_s : \gamma \in \gamma_s.\texttt{modeledSupertypes}()$

Modeled generalizations can be filtered for those the subject clabject is a subtype or supertype in.

$\gamma.\texttt{modeledGeneralizationsAsSubtype}() := \xi : \gamma \in \xi.\mathsf{subtype}$

$\gamma.\texttt{modeledGeneralizationsAsSupertype}() := \xi : \gamma \in \xi.\mathsf{supertype}$

Definition 9 (Model children)**:** The elements in a model can be separated into their linguistic types by intersecting the children with the set of all

linguistic types:

$\Sigma.\mathsf{generalizations} := \Sigma.\mathsf{children} \cap \mathsf{Generalization}$

$\Sigma.\mathsf{classifications} := \Sigma.\mathsf{children} \cap \mathsf{Classification}$

$\Sigma.\mathsf{inversions} := \Sigma.\mathsf{children} \cap \mathsf{Inverse}$

$\Sigma.\mathsf{equalities} := \Sigma.\mathsf{children} \cap \mathsf{Equality}$

$\Sigma.\mathsf{complements} := \Sigma.\mathsf{children} \cap \mathsf{Complement}$

For clabjects the case is a bit more complex since clabjects can recursively contain themselves. With operation 2, the clabjects in a model can be

Operation 2 $\Sigma.\texttt{clabjects}()$

$result \leftarrow \emptyset$
$queue \leftarrow \mathsf{Queue}(\Sigma.\mathsf{children} \cap \mathsf{Clabject})$
while $|queue| > 0$ **do**
  $current \leftarrow queue.\mathsf{pop}()$
  $result \leftarrow result \cup \{current\}$
  $queue.\mathsf{enqueue}(current.\mathsf{children} \cap \mathsf{Clabject})$
return $result$

further separated:

$\Sigma.\texttt{entities}() := \Sigma.\texttt{clabjects}() \cap \mathsf{Entity}$

$\Sigma.\texttt{connections}() := \Sigma.\texttt{clabjects}() \cap \mathsf{Connection}$

Through the generalizations, the clabjects inside a model form an inheritance hierarchy. As there is no circular subtyping, the hierarchies form directed acyclic graphs. A model offers an operation to retrieve the roots of these trees, i.e. the clabjects without supertypes.

$\Sigma.\texttt{inheritanceRoots}() :=$
  $\{\gamma \in \Sigma.\texttt{clabjects}() : \gamma.\texttt{modeledSupertypes}() = \emptyset\}$

To access the connection participations in one ontological level a model offers an operation to retrieve all roles from all the connections.

$\Sigma.\texttt{roles}() := \{\delta.\texttt{roles}() \forall \delta \in \Sigma.\texttt{connections}()\}$

Definition 10 (Ownership)**:** The owner of a feature is the clabject containing it.

$\eta.\texttt{owner}() := \gamma : \eta \in \gamma.\mathsf{children}$

The model of a clabject is again determined by the level.

$\gamma.\texttt{model}() := \Sigma_i : i = \gamma.\mathsf{level}$

The owner of a Correlation is the model containing the element. The model could also be determined by the level of the clabjects the Correlation connects.

$\lambda.\texttt{owner}() := \Sigma : \lambda \in \Sigma.\mathsf{children}$

Definition 11 (Ontological Level)**:** The level of a model is determined by the contained clabjects.

$\Sigma.\texttt{level}() := \gamma.\mathsf{level} : \gamma \in \Sigma.\mathsf{children}$

Each Correlation determines its level from the connected clabjects.

$\phi.\texttt{level}() := \phi.\mathsf{instance.level}$
$\xi.\texttt{level}() := \gamma.level : \gamma \in (\xi.\mathsf{subtype} \cup \xi.\mathsf{supertype})$
$\upsilon.\texttt{level}() := \upsilon.\mathsf{base.level}$

Definition 12 (Clabject Features)**:** Each clabject gathers its eigenFeatures from the contained elements. The *eigenFeatures* comprise all features that are directly contained in the clabject (and not inherited from supertypes or recursively included in a contained child clabject).

$\gamma.\texttt{eigenFeatures}() := \{\eta \in \mathsf{Feature} : \eta \in \gamma.\mathsf{children}\}$

From its supertypes, a clabject can gather the features it inherits (Operation 3) and finally all the features it offers. Among these features, the clabject offers an operation to search for a feature by name.

Operation 3 $\gamma.\texttt{modeledFeatures}()$

```
result ← ∅
for γ_s ∈ γ.modeledSupertypes() do
    result ← result ∪ γ_s.eigenFeatures()
return result
```

$\gamma.\texttt{features}() := \gamma.\texttt{eigenFeatures}() \cup \gamma.\texttt{modeledFeatures}()$
$\gamma.\texttt{feature}(name) := \eta \in \gamma.\texttt{features}() : \eta.\mathsf{name} = name$

The features of a clabject can then be further divided into attributes and methods:

$\gamma.\texttt{attributes}() := \{\eta \in \gamma.\texttt{features}() : \eta \in \mathsf{ATTRIBUTE}\}$
$\gamma.\texttt{methods}() := \{\eta \in \gamma.\texttt{features}() : \eta \in \mathsf{METHOD}\}$

Definition 13 (Connection Operations)**:** The order of a connection is the number of roles the connection defines.

$\delta.\texttt{order}() := |\{\psi \in \mathsf{Role} : \psi.\mathsf{connection} = \delta\}|$

A connection offers the roles it defines and the clabjects that participate in the connection as the destination of a role. Note that one clabject can participate in a connection through more than one role, so the number of roles does not need to match the number of participants. The full set of the connection's roles is comprised of its eigenRoles (i.e. roles not inherited, but defined directly) and the roles it inherits from supertypes, where the roleName is the key for overriding. Operation 4 gives the formal definition. If a connection has more than one direct supertype, there is no information to judge which role to chose.

Operation 4 $\delta.\texttt{roles}()$

$roles \leftarrow \delta.\texttt{eigenRoles}()$
for $\delta_s \in \xi.\mathsf{supertype} : \delta \in \xi.\mathsf{subtype}$ **do**
  for $\psi_s \in \delta_s.\texttt{roles}()$ **do**
    // recursive call to get the facade roles of the supertype
    if $\nexists \psi_i \in roles : \psi_i.\mathsf{roleName} = \psi_s.\mathsf{roleName}$ **then**
      $roles \leftarrow roles \cup \{\psi_s\}$
return *roles*

$\delta.\texttt{eigenRoles}() := \{\psi \in \mathsf{Role} : \psi.\mathsf{connection} = \delta\}$
$\delta.\texttt{participants}() := \{\psi.\mathsf{destination} : \psi \in \delta.\texttt{roles}()\}$

The target defined by one connection and one roleName is given by

$\delta.\texttt{navigate}(rN) := \psi.\mathsf{destination} : \psi \in \delta.\texttt{roles}() \wedge \psi.\mathsf{roleName} = rN$

and all the roleNames the connection offers are given by

$\delta.\texttt{roleNames}() := \{\psi.\mathsf{roleName} : \psi \in \delta.\texttt{roles}()\}$

and access to one role identified by the roleName is given by

$\delta.\texttt{role}(rN) := \psi \in \delta.\texttt{roles}() : \psi.\mathsf{roleName} = rN$

The navigable domain of a connection is the set of clabjects reachable through the connection. The domain is agnostic to any particular participant. The boolean parameter indicates whether or not the domain shall be limited to the navigable ones.

$\delta.\texttt{domain}(navigable) :=$
$\psi.\mathsf{destination} : \psi.\mathsf{connection} = \delta \wedge navigable \implies \psi.\mathsf{navigable}$

Since a clabject may participate in a connection more than once and the roleNames are unique, the return type of the operation is a set of roleNames rather than a single one.

$\delta.\texttt{roleNamesForParticipant}(\gamma) :=$
$\psi.\mathsf{roleName} : \psi.\mathsf{connection} = \delta \wedge \psi.\mathsf{destination} = \gamma$

he multiplicities are stored in the roles so the operation needs a roleName as parameter to identify the role with.

$\delta.\texttt{lower}(roleName) :=$
$\psi.\mathsf{lower} : \psi.\mathsf{connection} = \delta \wedge \psi.\mathsf{roleName} = roleName$

The upper multiplicity is retrieved similar to the lower.

$\delta.\texttt{upper}(roleName) :=$
$\psi.\mathsf{upper} : \psi.\mathsf{connection} = \delta \wedge \psi.\mathsf{roleName} = roleName$

As with multiplicity, the navigability is stored in the roles and the operation needs a roleName to identify the role by.

$\delta.\texttt{navigable}(roleName) :=$
$\psi.\mathsf{navigable} : \psi.\mathsf{connection} = \delta \wedge \psi.\mathsf{roleName} = roleName$

If the user does not input a roleName for a role (it is not mandatory), a default roleName is constructed for display and access from the destination clabject. The construction is performed according to the UML and OCL convention. The name of the destination is the default roleName with a lowercase first letter. If this roleName conflicts with another roleName in the connection (violating roleName uniqueness) the user is responsible for solving the conflict as the engine does not have enough information to

solve it on its own. The check for a default roleName checks if a value is set:

ψ.isDefaultRoleName() := $\exists\psi$.roleName

Definition 14 (Clabject Navigation)**:** The navigation roles of a clabject are the roles the clabject can navigate by. A clabject can navigate via a role if the connection has another role the clabject is the destination of (Operation 5). The navigations a clabject inherits from its supertypes are simply

Operation 5 γ.eigenNavigations()

```
result ← ∅
for δ ∈ Connection : δ.level = γ.level do
  if |{ψ ∈ δ.roles() : ψ.destination = γ}| > 0 then
    // γ participates in δ
    if |{ψ ∈ δ.roles() : ψ.destination = γ}| = 1 then
      // one destination, so there is no self reference
      result ← result ∪ {ψ ∈ δ.roles : ψ.destination ≠ γ}
    else
      // more than one destination, so there is a self reference
      result ← result∪
        {ψ ∈ δ.roles : ψ.destination ≠ γ ∨ ψ.navigable = True}
return result
```

collected. The overriding of inherited navigations for a clabject is discussed in 11.4.1.

γ.modeledNavigations() :=
  {γ_s.eigenNavigations() : $\gamma_s \in \gamma$.modeledSupertypes()}

The roles of a clabject, i.e. the navigations possible from the clabject is given by the union of the eigen- and inherited roles.

γ.navigations() := γ.eigenNavigations() $\cup$ γ.modeledNavigations()

The eigenConnections of a clabject are the connections which define a role the clabject is the target of:

γ.eigenConnections() := {ψ.connection$\forall\psi \in$ Role : ψ.destination = γ}

The model connections are the connections the clabject inherits from supertypes. More precisely, it is not the connection that is inherited, the role with the supertype as destination is inherited.

$\gamma.\texttt{modeledConnections}() :=$
$\{\gamma_s.\texttt{eigenConnections}()\forall\gamma_s \in \gamma.\texttt{modeledSupertypes}()\}$

Finally, the connections of a clabject are the union of the eigen- and inherited ones.

$\gamma.\texttt{connections}() := \gamma.\texttt{eigenConnections}() \cup \gamma.\texttt{modeledConnections}()$

With both navigations and connections, overriding in the most general case is not trivial. See 11.4.1 for details.

As the key for navigation from the view of the navigating clabject is the roleName to navigate by, the roleNames any clabject can navigate by are given by

$\gamma.\texttt{navRoleNames}() := \{\psi.\mathsf{roleName}\forall\psi \in \gamma.\texttt{roles}() : \psi.\mathsf{navigable} = True\}$

An actual navigation is defined by the traversal of the involved connection towards one role, yielding its destination as a result. The roleName is unique for one connection, but any source clabject may participate in more connections, thus having more than one navigation for the same roleName. So the result of one navigation is not one single clabject, but a set of clabjects.

$\gamma.\texttt{nav}(rN) := \{\psi.\mathsf{destination}\forall\psi \in \gamma.\texttt{navigations}() : (\psi.\mathsf{navigable} = True\wedge$
$\psi.\mathsf{roleName} = rN)\}$

The domain of a clabject in a connection are the destinations the clabject can navigate to via the connection. Of course, only a participating clabject can have a domain in a connection, $\gamma \in \delta.\texttt{participants}()$.

$\gamma.\texttt{domain}(\delta) := \psi.\mathsf{destination} : \psi.\mathsf{connection} = \delta \wedge \psi.\mathsf{navigable}\wedge$
$(\psi.\mathsf{destination} \neq \gamma \vee \exists\psi' : \psi \neq \psi' \wedge \psi'.\mathsf{destination} = \gamma)$

If a clabject participates more than once in a connection, it itself can be a part of the domain, exactly if one of the participations is navigable.

Definition 15 (Multiplicity Values)**:** A multiplicity is a tuple

$\mu := (lower, upper), lower \in \mathbb{N}_0, upper \in \mathbb{N}_0 \cup \{*\}$

where lower is the *lower* multiplicity bound and *upper* is the upper multiplicity bound. Multiplicity is normally displayed as $lower..upper$, except for the special cases

- $lower = upper$. multiplicity is displayed as *lower*
- $lower = 0 \wedge upper = *$. multiplicity is displayed as $*$

Multiplicity defines an operation to check the conformance of a single value to the constraints imposed by the multiplicity:

$\mu \times Integer \rightarrow \{True, False\}, (\mathtt{m}, i) \rightarrow lower \leq i \wedge (upper \geq i \vee upper = *)$

Definition 16 (Modeled Classification)**:** As with generalizations, classifications also define a tree structure and any clabject can retrieve the types and instances that are defined by the existence of classification relationships. With operation 6 it is easy to define the operation for modeled instances by

Operation 6 γ.`modeledTypes`()

```
result ← ∅
queue ← Queue({γ} ∪ γ.modeledSupertypes())
while |queue| > 0 do
  current ← queue.pop()
  for φ ∈ Σ_γ.level.classifications do
    if φ.instance = current then
      result ← result ∪ (φ.type ∪ φ.type.modeledSupertypes())
return result
```

inverting the operation.

γ.`modeledInstances`() := $\{\gamma' \in \Sigma_{\gamma.\mathsf{level}+1} : \gamma \in \gamma'.\mathtt{modeledTypes}()\}$

γ.`isModeledType`(γ_i) := $\gamma \in \gamma_i.\mathtt{modeledTypes}()$

As a special case of these operations a clabject can not only retrieve the target clabjects, but also the classification relationships realizing them:

γ.`modeledClassificationsAsInstance`() := $\{\phi \in \Sigma_{\gamma.\mathsf{level}}.\mathsf{classification} : \gamma = \phi.\mathsf{instance}\}$

$\gamma.\texttt{modeledClassificationsAsType}() := \{\phi \in \Sigma_{\gamma.\mathsf{level}+1}.\mathsf{classification} : \gamma = \phi.\mathsf{type}\}$

Taking the modeled classifications into consideration (and nothing else), the clabject may retrieve its modeled types and instances.

$\gamma.\texttt{isModeledInstance}(\gamma_t) := \gamma_t \in \gamma.\texttt{modeledTypes}()$

Although the different notions of types and instances will not be defined until definition 30, the clabject can retrieve them from the model. Operations 7 and 8 give the definition of the operations for incomplete types. Of course, the incomplete types are also obtained by the expression $\gamma_t \in$

Operation 7 $\gamma.\texttt{completeModeledTypes}()$

```
result ← ∅
sources ←
    {γ} ∪ {γ_s ∈ γ.modeledSupertypes() : γ.isShallowSubtype(γ_s)}
for φ : φ.instance ∈ sources ∧ φ.kind ∈ {instantiation, isonym} do
    result ← result ∪ {φ.type}
    result ← result∪
        {γ_s ∈ φ.type.modeledSupertypes() : φ.type.isShallowSubtype(γ_s)}
return result
```

$\Sigma_{\gamma.\mathsf{level}-1}.\texttt{clabjects}() : \gamma.\texttt{isHyponym}(\gamma_t)$ but the incomplete*Model*Types operation cannot rely on reasoning to get its results but only on modeled information. That means there *has* to be a classification element between an instance and a type. Operation 8 gives the formal definition.

The blueprint of a clabject can only be indicated through a classification. Blueprint information is not inherited and there can only be one blueprint of a clabject.

$\gamma.\texttt{blueprint}() := \phi.\mathsf{type} : \phi.\mathsf{instace} = \gamma \wedge \phi.\mathsf{kind} = \mathsf{instantiation}$

The offspring of a clabject are the instances that have been created from that clabject as a blueprint. As blueprint information can only be within a classification, model offspring relies on classifications as well.

$\gamma.\texttt{modeledOffspring}() := \phi.\mathsf{instance} : \phi.\mathsf{type} = \gamma \wedge \phi.\mathsf{kind} = \mathsf{instantiation}$

Operation 8 γ.`incompleteModeledTypes`()

$result \leftarrow \emptyset$
$unsure \leftarrow$
$\{\gamma\} \cup \{\gamma_s \in \gamma.\texttt{modeledSupertypes}() : \gamma.\texttt{isShallowSubtype}(\gamma_s)\}$ // the classification sources which are not yet sure to be incomplete
$sure \leftarrow \{\gamma_s \in \gamma.\texttt{modeledSupertypes}() : \neg\gamma.\texttt{isShallowSubtype}(\gamma_s)\}$ // the ones which are already sure to be incomplete
for $\gamma_{sure} \in sure$ **do**
  $result \leftarrow result \cup \gamma_{sure}.\texttt{modeledTypes}()$ // all types are incomplete
for $\gamma_{unsure} \in unsure$ **do**
  for $\phi \in \gamma_{unsure}.\texttt{modeledClassificationsAsInstance}() : \phi.\mathsf{kind} = \mathsf{hyponym}$ **do**
    $result \leftarrow result \cup \{\phi.\mathsf{type}\}$
    $result \leftarrow result \cup \phi.\mathsf{type}.\texttt{modeledSupertypes}()$
    $unsure \leftarrow unsure \setminus \{\gamma_{unsure}\}$ // γ_{unsure} is no longer unsure and also processed
// the clabjects remaining unsure have to find their property on the type level
for $\gamma_u \in unsure$ **do**
  for $\phi \in \gamma_u.\texttt{modeledClassificationAsInstance}$ **do**
    // $\phi.\mathsf{type}$ itself cannot be incomplete or γ_u would not be unsure
    for $\gamma_t \in \phi.\mathsf{type}.\texttt{modeledSupertypes}()$ **do**
      if $\neg\phi.\mathsf{type}.\texttt{isShallowSubtype}(\gamma_t)$ **then**
        $result \leftarrow result \cup \{\gamma_t\}$
return *unsure*

kind	instance	type
instance	instance	type
isonym	isonym	complete type
hyponym	hyponym	incomplete type
instantiation	offspring	blueprint

Table 4.3: type and instance roles depending on the kind of a classification

Definition 17 (Classification Role)**:** Based on the kind of classification, the type and instance play a distinct role in the relationship. This role can be used to characterize the kind of the classification. For the navigation of the classification element itself, the roles *type, instance* remain valid as they are unique and not subject to change. Table 4.3 shows the classification kinds and the respective roles.

4.6 Relationship between Potency and Durability

A clabject's potency and a feature's durability do not constrain each other. The potency limits the overall depth of a clabject's isonym tree while the durability defines how deep in that tree the feature will be present (i.e. will endure). Thus, if the durability is smaller than the potency, there will be valid isonyms without that feature somewhere down the tree. If the durability exceeds the potency it means there can be instances of leafs of the isonym tree which will have a conforming feature. The reason to allow such constructs is that subtypes of a clabject may have a higher potency and therefore carry the inherited feature further down their own isonym tree than the supertype does. Figure 4.3 shows a schematic view of an attribute that is present in more than one isonym tree. There is however a relation between the potencies of subtypes and their direct supertypes. If a subtype does not add any feature with durability > 0 or participate in a connection with potency > 0, the potency of the sub-

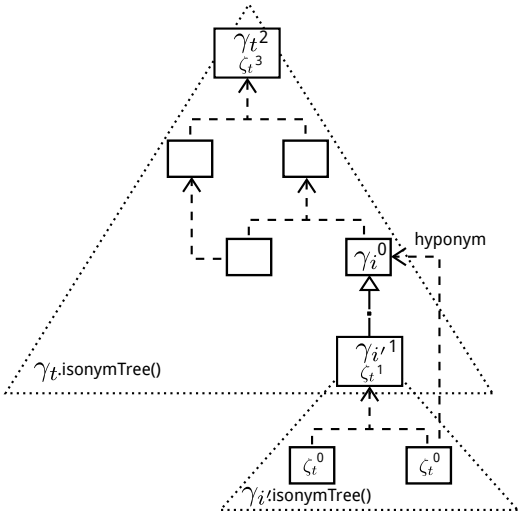


Figure 4.3: The attribute ζ_t lives longer than its owning clabject γ_t.

type and the supertype have to be equal. The reason is that any isonym of the supertype will also be an isonym of the subtype and vice versa. They share the same isonym tree and therefore their potency has to be the same.

$$
\begin{aligned}
&(\{\eta \in \gamma.\texttt{eigenFeatures}() : \eta.\mathsf{durability} > 0\} = \emptyset) \wedge \\
&(\{\delta \in \gamma.\texttt{eigenConnections}() : \delta.\mathsf{potency} > 0\} = \emptyset) \implies \\
&\quad (\forall \gamma_s \in \{\xi.\mathsf{supertype} \forall \xi : \gamma \in \xi.\mathsf{subtype}\} : \\
&\qquad \gamma.\mathsf{potency} = \gamma_s.\mathsf{potency})
\end{aligned}
$$

4.7 Connection Semantics

Connections have traits that impose constraints on the shape and existence of their instances. This section focusses on the formal consequences of these traits. As each of the three traits (transitivity, symmetry and reflexiveness) can have three states (true, false and not defined), setting a trait to false is just a strict statement as setting it to true. Only undefined traits do not impose any constraints on the classified domain.

4.7.1 Transitivity

If a connection is transitive, a connection chain a.b.c implies the existence of a third instance connecting a to c directly. If the transitivity statement is negated (i.e. if the not transitive statement is made) then such an instance cannot exist.

Operation 9 Connection Transitivity. The algorithm assumes the transitivity of the input connection δ is defined to be either true or false. The case that there is no statement on transitivity is not covered.

$domain \leftarrow \{\delta_i \in \Sigma_{\delta.\mathsf{level}+1}.\texttt{connections}() : \delta_i.\texttt{isInstance}(\delta)\}$
$shortcuts \leftarrow \emptyset$
for $\delta_i \in domain$ **do**
  // detect the chains starting from δ_i
  for $\psi \in \delta_i.\texttt{roles}() : \psi.\mathsf{navigable}$ **do**
    // the start of a chain has to be navigable
    $\gamma \leftarrow \psi.\mathsf{destination}$ // γ is the possible man in the middle
    for $\delta_{i'} \in (\gamma.\texttt{connections} \cap domain)$ **do**
      // γ takes part in two instances
      if $\exists \psi' \in \delta_{i'}.\texttt{roles}() : \psi'.\mathsf{destination} \neq \gamma \wedge \psi'.\mathsf{roleName} = \psi.\mathsf{roleName}$ **then**
        // γ is on the right end, so γ is the man in the middle
        $start \leftarrow (\psi'' \in \delta_i.\texttt{roles}() : \psi'' \neq \psi).\mathsf{destination}$ // the start is the other end of δ_i
        $shortcuts \leftarrow shortcuts \cup \{(start, \psi'.\mathsf{destination}, \psi.\mathsf{roleName})\}$
if $\delta.\mathsf{transitive}$ **then**
  for $(start, end, rN) \in shortcuts$ **do**
    $\exists \delta_i \in domain : \{start, end\} \subset \delta_i.\texttt{participants}() \wedge$
      $\psi \in (\delta_i.\texttt{roles}() : \psi.\mathsf{roleName} = rN).\mathsf{destination} = end$
else
  $\forall (start, end, rN) \in shortcuts :$
    $\nexists \delta_i \in domain : \{start, end\} \subset \delta_i.\texttt{participants}() \wedge$
      $\psi \in (\delta_i.\texttt{roles}() : \psi.\mathsf{roleName} = rN).\mathsf{destination} = end$

The algorithm searches for chains of three consecutive connected clabjects. The roles must be navigable and connect to the clabjects with the same roleName. The domain for this search are the instances of the connection under discussion. For each member of such chains, a shortcut connection from the start to the end (i.e. a direct connection with an equal navigable role) has to exist. If the connection states it is not transitive, such a connection must not exist.

4.7.2 Reflexiveness

A reflexive connection is a connection where the existence of an instance implies the existence of another instance connecting the participant to itself. A connection can only be reflexive if its order is two. Reflexiveness only applies to navigable roles and although its semantics remain intact for an undirected connection (both roles are navigable), it makes most sense for directed connections (only one role navigable). The domain for the algorithm is the same as for transitivity.

Operation 10 Connection Reflexiveness. The algorithm assumes the reflexiveness of the input connection δ is defined to be either true or false. The case that there is no statement about it is not covered.

$domain \leftarrow \{\delta_i \in \Sigma_{\delta.\mathsf{level}+1}.\texttt{connections}() : \delta_i.\texttt{isInstance}(\delta)\}$
for $\delta_i \in domain$ **do**
  for $\psi \in \delta_i.\texttt{roles}() : \psi.\mathsf{navigable}$ **do**
    $\psi' \leftarrow \psi'' \in \delta_i.\texttt{roles}() : \psi'' \neq \psi$
    $\delta.\mathsf{reflexive} = \textbf{true} \iff$
      $\exists \delta' \in domain : \delta'.\texttt{participants} = \{\psi.\mathsf{destination}\} \wedge$
      $\delta'.\texttt{role}(\psi'.\mathsf{roleName}).\mathsf{navigable} = \textbf{true}$

4.7.3 Symmetry

A symmetric connection is a connection where each navigation works in both ways. Only binary (order two) connections can be symmetric. Every

clabject reachable through a symmetric role is also able to reach the other clabject via the same roleName (through another instance of the symmetric role). The algorithm operates in the same way as the algorithm for reflex-

Operation 11 Connection Symmetry. The algorithm assumes the symmetry of the input connection δ is defined to be either true or false. The case that there is no statement on it is not covered.

$domain \leftarrow \{\delta_i \in \Sigma_{\delta.\mathsf{level}+1}.\texttt{connections}() : \delta_i.\texttt{isInstance}(\delta)\}$
for $\delta_i \in domain$ **do**
  for $\psi \in \delta_i.\texttt{roles}() : \psi.\mathsf{navigable}$ **do**
    $\psi' \leftarrow \psi'' \in \delta_i.\texttt{roles}() : \psi'' \neq \psi$
    $\delta.\mathsf{symmetric} = \textbf{true} \iff \exists \delta' \in domain :$
      $\delta'.\texttt{participants}() = \delta_i.\texttt{participants}() \wedge$
      $\delta'.\texttt{role}(\psi.\mathsf{roleName}).\mathsf{navigable} \wedge$
      $\delta'.\texttt{role}(\psi.\mathsf{roleName}).\mathsf{destination} = \psi'.\mathsf{destination}$

iveness. For each navigable role of an instance connection there has to exist another instance realizing the same navigation in the opposite direction.

4.8 Correlation Semantics

Correlations are either set relationships, classifications or generalizations. Classification semantics are covered in chapter 6 and 39 while generalizations are addressed in 40. They are both very important and too complex to be dealt with here. The `isInstance` operation is based on the definitions presented in this section is also introduced in chapter 6.

4.8.1 Equality

Two clabjects are equal if there is no way to distinguish them apart from their name. In other words, they are different names for the same thing and any instance of one will also be an instance of the other and vice versa. There cannot be a clabject that is an instance of one but not the other.

$$(\gamma, \gamma' : \exists \upsilon_e : \gamma = \upsilon_e.\mathsf{base} \wedge \gamma' = \upsilon_e.\mathsf{equal} \vee \gamma' = \upsilon_e.\mathsf{base} \wedge \gamma = \upsilon_e.\mathsf{equal})$$
$$\implies \forall \gamma_i \in \mathsf{CLABJECT} : \gamma_i.\mathtt{isInstance}(\gamma) \iff \gamma_i.\mathtt{isInstance}(\gamma')$$

Equality exists whenever the sets defined by two clabjects are the same. There need not be an expressed equality relationships between them, but if there is one, the two clabjects must be equal. Equality is a symmetric property.

4.8.2 Inversion

If a connection is the inverse of another it inverses all the navigations the first one introduces. So any clabject reachable from an instance of one clabject can navigate to that clabject via an instance of the inverse connection. Only binary connections can be connected by inversions.

$$(\delta, \delta' : \exists \upsilon_i : \delta = \upsilon_i.\mathsf{base} \wedge \delta' = \upsilon_i.\mathsf{inverse}) \implies$$
$$\forall \delta_i : \delta_i.\mathtt{isInstance}(\delta) :$$
$$\forall \psi_i \in \delta_i.\mathtt{roles}() : \psi_i.\mathsf{navigable} = true :$$
$$\exists \psi' : \psi'.\mathsf{connection}.\mathtt{isInstance}(\delta') \wedge$$
$$\psi'.\mathsf{navigable} = true \wedge \psi'.\mathsf{destination} \neq \psi_i.\mathsf{destination} \wedge$$
$$\psi_i.\mathsf{destination} \in \delta_i.\mathtt{participants}()$$

4.8.3 Complement

A complement is a statement about three clabjects. The third ingredient is the universe in which the complement is defined. So if a complement exists, every instance of the universe has to be either an instance of the complement or the original clabject, not both and not none. Also, there can be no instance of either the base or the complement that is not an instance of the universe.

$$\upsilon_c \implies \forall \gamma_i : \gamma_i.\mathtt{isInstance}(\upsilon_c.\mathsf{universe}) \iff$$
$$(\gamma_i.\mathtt{isInstance}(\upsilon_c.\mathsf{base}) \iff \neg \gamma_i.\mathtt{isInstance}(\upsilon_c.\mathsf{complement}))$$

4.9 Overview of Operations & Default Values

This section contains a table of the previously defined operations as well as a table listing the default values of the PLM traits.

Table 4.4: PLM Operations

Element	Operation	Return Type	Result
Element	linguisticType	PLM Type	the linguistic type of the element
Ontology	isWellFormed	Boolean	true if the ontology is well formed
Model	level	Integer	the level of the model is the level of the contained clabjects
Model	classifiedModel	Model	the model classified by the called model
Model	classifyingModel	Model	the model classifying the called model
Model	clabjects	Set(Clabject)	all the clabjects contained (recursively) in the model
Model	entities	Set(Entity)	the entities of the clabjects()
Model	connections	Set (Connection)	the connections of the clabjects()
Model	inheritanceRoots	Set(Clabject)	the clabjects in the model without a model supertpye
Model	roles	Set(Role)	the roles contained in the connections of the model
Model	isWellFormed	Boolean	true if the model is well formed
Model	isRootModel	Boolean	true if there are no models above

Model	isLeafModel	Boolean	true if there are no models below
Feature	owner	Clabject	the clabject owning the feature
Feature	isWellFormed	Boolean	true if the feature is well formed
Attribute	isWellFormed	Boolean	true if the attribute is well formed
Correlation	owner	Integer	the model owning the correlation
Classification	level	Integer	the level of the owning model
Classification	isWellFormed	Boolean	true if the classification is well formed
Generalization	level	Integer	the level of the owning model
Generalization	isWellFormed	Boolean	true if the generalization is well formed
SetRelationship	level	Integer	the level of the owning model
SetRelationship	isWellFormed	Boolean	true if the set relationship is well formed
Clabject	modeledSupertypes	Set(Clabject)	the clabjects reachable via supertype generalizations
Clabject	modeledSubtypes	Set(Clabject)	the clabjects reachable via subtype generalizations
Clabject	modeledGeneralizationsAsSubtype	Set (Generalization)	the generlalizations the clabject is the subtype of
Clabject	modeledGeneralizationsAsSupertype	Set (Generalization)	the generlalizations the clabject is the supertype of

Clabject	eigenNavigations	Set(Role)	the roles the clabject is the destination of
Clabject	modeledNavigations	Set(Role)	the roles inherited from supertypes
Clabject	navigations	Set(Role)	the roles the called clabject can reach another one with
Clabject	eigenConnections	Set (Connection)	the connections the called clabject participates in
Clabject	modeledConnections	Set (Connection)	the connections inherited from supertypes
Clabject	connections	Set (Connection)	the connections the called clabject can navigate by
Clabject	navRoleNames		the roleNames the called clabject can reach others with
Clabject	nav(roleName)	Set(Clabject)	the clabjects reachable from the called clabject via the roleName
Clabject	domain(Connection)	Set(Clabject)	the clabjects reachable from the called clabject via the connection
Clabject	features	Set(Feature)	all the features in the facade of the called clabject
Clabject	attributes	Set(Attribute)	all the attributes in the facade of the called clabject
Clabject	methods	Set(Method)	all the methods in the facade of the called clabject

Clabject	model	Model	the model the clabject is recursively contained in
Clabject	eigenFeatures	Set(Feature)	the features defined directly by the clabject
Clabject	modeledFeatures	Set(Feature)	the features inherited from supertypes
Clabject	feature(name)	Feature	the feature with the provided name
Clabject	modeledInstances	Set(Clabject)	the clabjects that are either directly connected to the called clabject or one of its subtypes with a classification or are a subtype of a directly connected one
Clabject	modeledTypes	Set(Clabject)	the inverse operation to modelInstances
Clabject	isModeledType (Clabject)	Boolean	true if the called clabject is a modeled type of the provided one
Clabject	isModeledInstance (Clabject)	Boolean	true if the called clabject is a modeled instance of the provided one
Clabject	modeledClassifi cationsAsInstance	Set (Classification)	the classifications the clabject is the instance of
Clabject	modeledClassifi cationsAsType	Set (Classification)	the classifications the clabject is the type of

Clabject	completeModeled Types	Set(Clabject)	the clabjects which are a complete modeled type of the called clabject
Clabject	incompleteModeled Types	Set(Clabject)	the clabjects which are an incomplete modeled type of the called clabject
Clabject	modeledOffspring	Set(Clabject)	the clabjects which the called clabject is the blueprint of
Clabject	blueprint	Clabject	the clabject the called clabject was constructed from
Clabject	isWellFormed	Boolean	true if the clabject is well formed
Connection	order	Integer	the number of roles in the connection
Connection	eigenRoles	Set(Role)	the roles directly contained in the connection
Connection	roles	Set(Role)	the roles defining the facade of the connection
Connection	participants	Set(Clabject)	the clabjects taking part in the connection
Connection	navigate(roleName)	Clabject	the clabject that participated in the called connection with the provided roleName
Connection	domain(navigable)	Set(Clabject)	the clabjects the connection connects. If navigable is true then only those which are reachable

Connection	roleNames	Set(String)	all the roleNames present in the connection
Connection	role(roleName)	Role	retrieves the role identified by roleName
Connection	roleNamesForParticipant(Clabject)	Set(String)	the roleNames the participant participates with
Connection	lower(roleName)	int	the lower multiplicity of the role identified by roleName
Connection	upper(roleName)	int	the upper multiplicity of the role identified by roleName
Connection	navigable(roleName)	Boolean	true if the role identified by the roleName is navigable
Connection	isWellFormed	Boolean	true if the connection is well formed
Role	isWellFormed	Boolean	true if the role is well formed
Role	isDefaultRoleName	Boolean	false if the user expressed a roleName

Table 4.5: PLM Default Values

Element	Trait	Default	Description
Element	expressed	true	an engine that created the element may change the value
Element	relevant	true	the vast majority of model elements should be relevant
Element	fix	false	fixing of model elements is only feasible for collaborative scenarios

Visualizer	durability	element durability or potency or 0	only elements involved in classification can have a durability and it can never be higher than the classification potency
Clabject	level	none	the level has to be entered by the user
Clabject	potency	number of existing classified models	per default the clabject potency is adjusted so that the clabject affects all the remaining levels
Role	navigable	true	navigation is enabled per default
Role	lower	0	per default the classified domain does not need to redefine the role
Role	upper	*	per default the classified domain is not limited in the redefinition of the role
Connection	transitive	not set	per default no statement on transitivity is given
Feature	durability	the clabject's potency	per default the feature lives as long as the clabject
Attribute	mutability	the feature's durability	per default the attribute value lives as long as the attribute
Generalization	intersection	not set	per default no statement on intersection is given
Generalization	disjoint	not set	per default no statement on disjointness is given

Generalization	complete	not set	per default no statement on completeness is given
Classification	kind	instantiation	the default classification usecase is instantiation

Chapter 5

The Level Agnostic Modeling Language

The following chapter introduces the Level Agnostic Modeling Language (LML)(16, 32) used to render multilevel models in the PLM. After the requirements leading to the chosen design are established, the features of the language are presented and discussed, followed by the definition of the rendering mechanics and a tour of the concrete syntax of the PLM elements. The chapter concludes by representing the PLM in the LML (an thus conceptually, using itself).

5.1 Requirements to a level agnostic modeling language

The UML was first released by the OMG in 1997 (53) and has been the subject of research and evolution ever since. Its concrete syntax for class diagrams has however changed very little. There have been some additions and tweaks over the years (e.g. stereotypes or powertypes) but the basics have stayed the same. This stability is a testimony to the success of the concrete syntax. A general purpose modeling language such as the LML

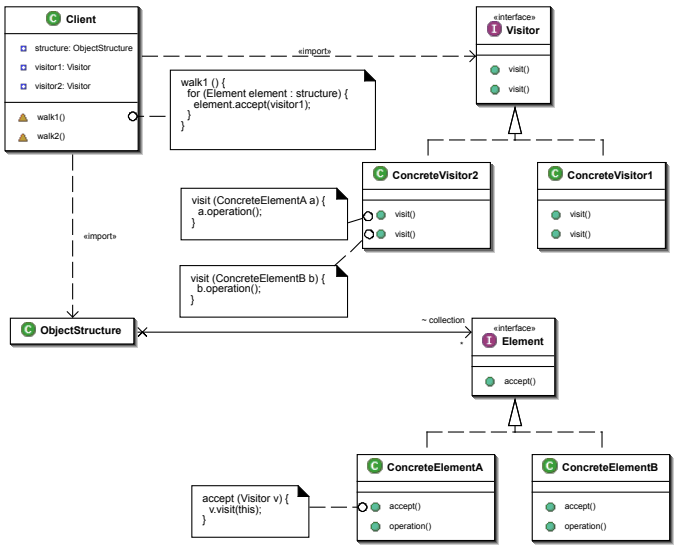


Figure 5.1: UML class diagram of the Visitor pattern.[2]

should therefore deviate from the UML visualization conventions to the smallest extent possible, and only when absolutely necssary.

5.1.1 UML Best Practices

The UML approach (figure 5.1) of representing entities as rectangles and the connections between them as lines has proven to be very intuitive even to users not familiar with modeling. The UML style of rendering properties inside the shape of the containing element not only visualizes the natural containment order into the model but also gives the model a clean and tidy look. Last but not least, some of the traits of UML elements such as navigability and multiplicity are displayed using concrete syntax pieces. While not full elements on their own, the inclusion of the traits as concrete syntax (e.g. the absence of an arrowhead if a connection is not navigable) not only displays the trait in a way that resembles its semantic meaning but gives the whole model more detail without adding more complexity. The LML tries to make use of these well-proven and established concepts.

5.1.2 Support for mainstream modeling paradigms

As shown in 3, there is quite a number of different approaches to modeling in general and to viewing model content in particular. The main identified paradigms are

- construction oriented top-down modeling with types where created first and then instances are created from them,
- data oriented design of only one ontological level (i.e. a type model (45)) with an explicit focus on attributes.
- analysis oriented modeling starting with a given set of elements and creating new information or revealing implied information.

The LML is designed in full knowledge of all three paradigms and aims to be able to support each of them.

5.1.3 Support reasoning services

Reasoning services need explicit support for correlations inside the model to be on the one hand able to access the information needed for their operation, and on the other hand to store their results. The LML tries to support reasoning services by providing model elements representing correlations and by offering rendering mechanisms for almost all the linguistic traits that might be interesting to show.

5.2 LML representation of PLM concepts

The LML provides a visual concrete syntax for every PLM concept. Most of the renderings reuse the well proven UML style. The details of the LML

[2]Credit to author Giacomo Ritucci published under Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) via Wikimedia Commons (`http://de.wikibooks.org/wiki/Datei:Visitor_UML_class_diagram.svg`)

syntax are covered in (16). Figure 5.5 renders the PLM metamodel using the LML syntax.

Ontology is the outermost visual container and in any LML diagram there is only one ontology. So an ontology is rendered as a rounded rectangle surrounding all the other elements. The ontologies name is displayed in the top right corner.

Models are the only children of ontologies and every model encapsulates one ontological level. So visually, models form the layers of the model stack inside the ontology. Hence a model is rendered as a horizontal layer inside the ontology rectangle. The model's name is displayed just below the horizontal line on the left hand side.

Entities represent the nodes in the model graph and are the visual counterparts to classes in the UML. So the visual notation reuses the well-known rectangles with the three compartments: The header compartment, the attribute compartment and the method compartment.

Connections realize the edges in the model graph, but are clabjects themselves. So in LML every connection is the visual equivalent to associationClasses in UML. Therefore a connection has a nodal shape as well. To distinguish connections from entities, the shape for a connection is a flattened hexagon. Inside the shape the compartments are the same as for entities. The representation of connections as edges in the model graph is discussed in the following section.

Roles visualize the participation of a clabject in a connection. Visually, they are lines connecting the participant to the connection. The traits are rendered in the same way as in the UML. If the role is navigable, the line at the participant ends with an arrowhead. The multiplicity constraint is displayed at the participant end as is the roleName.

Attributes are rendered in text inside the attribute compartment of the owning clabject. The rendered traits are limited to name, datatype, value, durability and mutability.

Methods are rendered in text inside the method compartment of the owning clabject. The textual representation is the method signature.

Generalizations are rendered as lines between the connected clabjects. At the supertype end, a triangle arrowhead indicates the supertype. A generalization can connect multiple subtypes or multiple supertypes. In that case the lines originate from a virtual center point.

Classifications are the only relationships crossing model boundaries. In order not to break the visual boundaries between models, they are rendered as dashed lines. Classifications are always directed from the instance to the type and the arrowhead at the type is an open triangle for visual distinction from generalizations. Roles can be displayed at each end, but their content depends on the kind of the classification.

SetReleationship renderings do not differ between the concrete subtypes. Each relationship is binary and connects two clabjects. The optional universe of a complement is shown in text form inside the shape. The shape at the meeting point of the lines connecting the ends is a rectangle with curved long sides. As set relationships do not have names the type of relation is shown inside the rectangle.

5.3 Main Innovations in the LML

The main enhancement of the LML in comparison to the UML is that LML is able to epxress the same information with a significantly lowered set of concepts. Additionally the LML provides several features to enhance the UML user experience and to foster multi-level modeling.

5.3.1 Dottability

Many of the PLM elements connect other elements at various levels of abstraction. The intuitive way to render them is with a line, but in some graph definitions, edges are not real elements in their own right but only tuples of the nodes they connect. In the LML connections do have more semantics than just the affected nodes, if they are given a shape of their own (like UML AssociationClasses) the natural paradigm of representing connections is broken and the shape is perceived as a node in the graph as well. To break this cycle every conecting PLM element with rich semantics can be rendered in a visually insignificant, "dotted", way to resembles edges in the model graph (see figure 5.2). If the focus is on their semantics as an element in themselves they can be rendered in an exploded form using a defined shape to display their content.

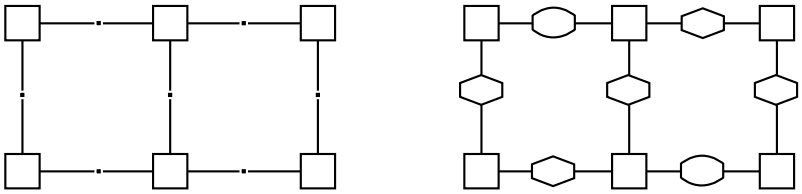


Figure 5.2: Example model graph, once dotted on the left hand side and with exploded rendering on the right hand side.

5.3.2 Clabject Header Compartment

In traditional modeling languages, the header compartment of entity symbols only contains the entity's identifier. This is arguably the most important trait of a classifier (and is omitted only for anonymous instances or to visualize a model purely as symbols), but over time the space in the header compartment became too precious to be wasted. Normally the identifier does not take up the whole horizontal space in an entity symbol since all the

features have to be rendered within the same bounds, and methods as well as fields render their own identifier along with other traits in the same line.

In the UML, it is feasible to append the type the clabject was constructed from with a preceding colon to the clabject's name. In the constructive modeling mode this mechanism would be a possibility to render the blueprint trait, but in exploratory mode the space would be lost. The general idea of showing correlations to the side of the identifier is a good principle, so clabject specific information should be shown with its symbols bounds.

5.3.2.1 Proximity Indication

With proximity indication it is possible to include generalization, classification and containment information in the header compartment. Every clabject involved is displayed with its identifier. To highlight the identifier of the clabject, it is rendered in bold font while the others are rendered in normal font. Viewing engines can opt to render the involved clabjects even with a smaller font according to user configuration.

Displayed proximity information is not limited to one level in the respective hierarchy as with the UML style used to show blueprints. As an upper bound, all the information in the clabject's supertype tree and type tree can potentially be included in the header compartment. Suppose that `A` is a clabject to be rendered.

generalization Any supertype `B` can be shown in the header compartment by prepending it to the identifier, separated by a less than sign: `B<A`. The less than sign is intended to resemble the arrow head of a generalization relation pointing towards the supertype. Subsequent supertypes can be added to the inheritance path, where `C<B<A` shows the clabject C as a supertype of both `A` and `B`.

To show the relationhsip between `A` and `C` without relying on `B`, double less than signs indicate the indirect existence of the relation (indirect

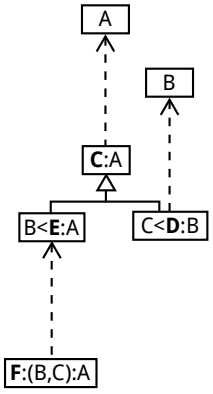


Figure 5.3: Exhaustive Proximity Indication Example

in the sense that it is valid but realized by the existence of another clabject): `C<<A`.

classification The same mechanism of showing inheritance can be used to show classification. The type is separated from the instance using a colon, resembling the UML style of displaying a type. Classification information is shown to the right of the identifier. Again the principle can be applied repeatedly covering multiple levels: `A:T:TT`.

containment Finally, containment is the last ingredient in a fully fleshed out header compartment. Containment forms the natural hierarchy of model elements. In the LML the owner is separated from the owned element by a dot, unlike the double colon in the UML. By applying the pattern recursively each element receives a unique navigation path or fullname from the root element. As containment is the closest relation, it is shown directly in front of the identifier but not in bold font: `B<Example.O2.A:T`

While it is theoretically possible to apply the proximity features to displayed identifiers other than the clabject's one, it is prohibited by the LML concrete syntax. Figure 5.3 shows an example of a verbose proximity indication.

5.3.2.2 Attribute Value Specification

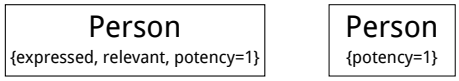
An attribute value specification (AVS) is a way to visualize traits of a clabject in a textual form in the header compartment. The concrete syntax is a comma separated list of `key=value` pairs where `key` is the name of the trait and `value` is the value of the trait. The list is enclosed by curly braces.

```
{key=value,key=value,..,key=value}
```

Semantically the AVS is a way for the user to either de-clutter the visual rendering of a clabject or emphasize a certain trait, perhaps one that does not have a distinct concrete syntax of its own (like *relevant* for example). By default, no traits are shown in the AVS. Potentially, the AVS can show all the traits, including the ones already rendered through special concrete syntaxes (like name, for example). The AVS is located underneath the string representing the identifier and proximity indication. It is rendered in smaller font. The rendering engine may opt to display it in another font (monospace has proven useful) to distinguish it from the artefact's name.

Boolean traits are a special case in AVS rendering. In the UML style of tagged values a boolean key is shown when it is true and not shown when it is false. The rendering saves horizontal space because the "=true" is omitted, but as the boolean key has to be shown if it is true the AVS has to include all positive keys. Given that there are booleans which default to true, the AVS becomes large by default and the user loses flexibility and control, because displaying the boolean traits is taken out of his hands. Figure 5.4 shows the disadvantage of the approach. Even though only the potency trait shall be rendered, the other ones have to be shown as well. To increase flexibility the LML does not apply the principle of strict boolean rendering. So if a boolean trait is not shown, it does not mean it is false. There is no statement implied. As the LML needs a way to show false boolean traits as well and the horizontal space shall be preserved as much as possible, a false trait is rendered with an exclamation mark in front of it.

Figure 5.4: AVS and boolean traits



5.3.3 Special Visual Notation for Traits

While all the traits of an element are usually accessible through the context menu or property panes of the modeling tool and the LML additionally offers the possibility to display any trait along with its value in the AVS, some traits are special enough to be given their own distinct concrete syntax. The goal was to make the rendering of traits as inconspicuous as possible. Whenever a trait has the default value, the assumption is that the trait does not have enough significance to be shown. So traits are only shown when their value differs from the default one. Whenever possible the conventions conform to the original ones of Atkinson and Kühne (9).

5.3.3.1 Potency

The potency of a clabject can be shown as a superscript after the clabject's identifier in the header compartment. Potency does not have a fixed default value, as there is no global "uninteresting" value for potency. In fact, the to-be-expected value of potency depends on the maximum number of levels the ontology is intended to have. A typical model element at L0 will have offspring all the way to L_{max} so the expected potency at L0 would be $_{max}$. However, since traditional modelling technologies support only two ontological levels and thus every type modeled had a potency of 1, the default potency value is 1.

durability and mutability Durability and mutability are also called feature potency and value potency, especially when talking about potency in general or in an informal way. The concrete syntax is the same as for potency. The value is shown as a superscript next to the identifier it specifies.

The default value for durability is the potency value of the owning clabject. So the durability is shown only when it differs from the clabjects potency, which makes sense as only then does the feature have a different lifespan to the clabject.

The default value for the mutability follows the same pattern. The default value is the features durability, so it is only shown when the mutability is different than the feature durability.

5.3.3.2 Level

The level trait is only relevant for clabjects. In normal viewing mode it is not necessary to show each individual clabject's level trait since this is usually evident from the model they are contained in. The level trait becomes important when slices of models or single elements are shown, especially if elements from different levels are shown in one view alongside each other. The level is then rendered as a subscript after the clabject identifier. The level position is the same as the potency's but it is shown as a subscript rather than a superscript. Since the editor can not determine when the level trait is of importance, the rendering of the level is controlled by user settings.

Figure 5.5: The Pan Level Metamodel represented with the PLM

5.4 Representing the PLM in PLM rendered with the LML

Now that the abstract and concrete syntax have been established, this final section shows how the PLM can be represented by itself using the LML as the concrete syntax. Although the semantics is the same as the UML version in (figure 4.2) the different visualization emphasizes different properties of the metamodel:

no complex attributes PLM Attributes are not able to express relationships between metamodel types. So for every relation between two PLM Elements there has to be a Connection. The motivation is the strict separation of data and element relations.

dotted notation In PLM, every Connection is displayed either in exploded or dotted notation. For the metamodel, every Connection looks like the edge in a graph and not like an AssociationClass, so they are all rendered dotted. The visual consequence are the little rectangles in the middle (or at junctions) of the line.

Compared with the UML-like representation, the most obvious change is that the inheritance hierarchy is not as visible anymore. The reason is that before the only lines were the Generalization lines and therefore the layout of the Elements could be hierarchical from top to bottom according to those Generalizations. With the additional lines in the picture, the modeler (or a layout engine) is presented with the task of optimizing the layout according to a some concrete criteria. Most commonly these criteria are number of line crossings, the sum of all line lengths (possibly weighted) or the size of the bounding rectangle. In the given figure, these different concerns were combined to optimize readability. The immediate benefit of the LML representation is the visualization of the relative importance of elements based on the number of edges entering or leaving them and their overall

size. The central role Clabject plays in the PLM becomes immediately visible through the many incoming Connections and the resulting size of the rectangle that represents it.

Chapter 6

Clabject Classification

One vital ingredient of a comprehensive multilevel modeling framework is a formal and complete definition of the semantics of classification. The definition has to be formal so they it be processed by reasoners, transformation engines or other kind of information processing systems, and it has to be complete so all necessary information needed by these systems to make appropriate inferences is available. This chapter presents such a formal and complete model of classification on top of the PLM. The first step is to analyse the local conformance of just the two clabjects involved in a classification relationship (type and instance). The next step then considers their connected clabjects as well until, finally, every mandatory connection is analysed. For connections there is more work to do than for entities. A connection can only be an instance of another connection if the connected clabjects conform as well. Additionally, multiplicity conformance has to be checked.

This chapter contains the needed layers of conformance definition, starting with the one without any dependencies (feature conformance) and gradually building up to the `isInstance` operation. The steps along the way are the local conformance of two clabjects, neighbourhood conformance and property conformance.

6.1 Feature Conformance

Definition 18 (Feature Conformance)**:** As the name is the primary key for the identification of a Feature within a clabject, the names have to match. The durability of a feature has to be one lower than the durability of the feature it conforms to. $*$ durabilities are conformed to by any durability.

$$\begin{aligned}&\eta.\texttt{conforms}(\eta') := (\eta.\mathsf{name} = \eta'.\mathsf{name}) \wedge \\ &\quad (\eta'.\mathsf{durability} \neq * \implies \eta.\mathsf{durability} = \eta'.\mathsf{durability} - 1) \wedge \\ &\quad (\eta' \in \mathsf{ATTRIBUTE} \implies \eta.\texttt{conforms}_a(\eta')\end{aligned}$$

Definition 19 (Attribute Conformance)**:** Given two attributes ζ, ζ',

$\zeta.conformsTo(\zeta')$

is true if ζ fulfils the definition of ζ' so that ζ could have been created from ζ'. *Being created from* means that the clabject of ζ' is instantiated and as part of this process, ζ' creates an attribute for the instance.

$$\begin{aligned}&\zeta.\texttt{conforms}_a(\zeta') := \\ &\quad (\zeta.\mathsf{datatype} = \zeta'.\mathsf{datatype}) \wedge \\ &\quad (\zeta'.\mathsf{mutability} \neq * \implies \zeta.\mathsf{mutability} = max[\zeta'.\mathsf{mutability} - 1, 0]) \wedge \\ &\quad ((\zeta'.\mathsf{mutability} = 0) \implies (\zeta.\mathsf{value} = \zeta'.\mathsf{value}))\end{aligned}$$

The semantics of mutability come into play when the mutability of the to-be conformed attribute is zero. In that case, the value of the Feature belongs to the type and cannot be changed by any of the instance clabjects. As a consequence, for the conformance check of the Features, the value has to be equal. As mutability can be lower than the durability, it is possible to conform to a mutability of zero. The conforming mutability then has to be zero as well. The operation $max[\dots]$ returns the maximum of the specified values.

A datatype is a valid type for a primitive data value. Examples include but are not limited to $String, Boolean, Integer, Float, Double, \dots$ A formal definition for these types is not necessary as they are given by the environment in which the model is implemented in. The semantics of a datatype

for a PLM model is the static ability to compare two attributes for conformance based on their intended datatype and secondly the ability to validate the well-formedness of an attribute by comparing the actual type of the value against its defined datatype. The conformance of datatypes is stated with the equals sign in the definition. In the implementing environment, there may be other possibilities than exact equality, for example bigInteger conforming to Integer.

Definition 20 (Method Conformance)**:** Methods consist of a body containing the execution information in textual form and the input and output parameters. As the runtime semantics and effects are not part of the PLM, but the concern of the implementing domain, the only static information that can be processed is the name and durability of the method. It is possible to judge the number (and name or even the types the expression evaluates to) of the parameters, but the conformance requirements vary with the implementing platform. So formally, method conformance is the same as feature conformance.

6.2 Local Conformance

Definition 21 (Clabject Local Conformance)**:** Local conformance for clabjects covers a clabject's traits and features. Roles cannot be a part of local conformance as the navigations possible from a clabject are defined by the connection and are therefore outside the scope of local conformance.

$$\gamma.\texttt{localConforms}_{clabject}(\gamma') := (\gamma.\mathsf{level} = \gamma'.\mathsf{level} + 1) \wedge$$
$$(\forall \eta' \in \gamma'.\texttt{features}() : \eta'.\mathsf{durability} > 0 : \exists \eta \in \gamma.\texttt{features}() :$$
$$\eta.\texttt{conforms}(\eta'))$$

Definition 22 (Entity Local Conformance)**:** Entities do not extend the linguistic definition of clabjects other than being a disjoint and complete subtype (together with connection). While this is important information is not relevant for local conformance, so the operation for entity local conformance delegates to clabject local conformance.

$\epsilon.\texttt{localConforms}(\epsilon') := \epsilon.localConforms_{clabject}(\epsilon')$

Definition 23 (Connection)**:** The main things connections add to clabjects is the definition of roles. Roles cannot exist without the defining connection, so they belong to the local domain of a connection. The links of the role to the destination are not part of the local domain of the connection as they touch another clabject. So only the trivial traits of the role can be checked.

$\delta.\texttt{localConforms}(\delta') := (\delta.\texttt{localConforms}_{clabject}(\delta'))\wedge$
$(\delta.\texttt{order}() = \delta'.\texttt{order}()) \wedge (\forall\psi' \in \delta'.\texttt{roles}() :$
$\quad \exists\psi \in \delta.\texttt{roles}() : ($
$\quad\quad \psi.\mathsf{roleName} = \psi'.\mathsf{roleName} \wedge \psi.\mathsf{navigable} = \psi'.\mathsf{navigable}))$

6.3 Neighbourhood Conformance

Neighbourhood conformance extends the scope of local conformance to the connections and destinations of the subjects. Since there should be no recursion at this stage, the property that needs to be checked at the navigation ends is local conformance.

Definition 24 (Clabject Neighbourhood Conformance)**:** Clabject neighbourhood conformance extends the scope of local conformance to the clabject's navigations (i.e. the roles) and the destinations reachable through its mandatory[1] roles. The scope is therefore extended by the immediate neighbourhood of the clabject.

$\gamma.\texttt{neighbourhoodConforms}_{clabject}(\gamma') := (\gamma.\texttt{localConforms}(\gamma')) \wedge ($
$\forall\psi' \in \gamma'.\texttt{navigations}() : (\psi'.\mathsf{connection.potency} > 0 \wedge \psi'.\mathsf{lower} > 0) :$
$\quad \exists\psi \in \gamma.\texttt{navigations}() :$
$\quad\quad (\psi'.\mathsf{roleName} = \psi.\mathsf{roleName}) \wedge (\psi'.\mathsf{navigable} = \psi.\mathsf{navigable})\wedge$
$\quad\quad (\psi.\mathsf{destination}.\texttt{localConforms}(\psi'))\wedge$
$\quad\quad (\psi.\mathsf{connection}.\texttt{localConforms}(\psi'.\mathsf{connection}))$

[1] Informally, a role is mandatory if the connection has potency greater than zero and any other of its roles has a lower multiplicity greater than zero.

Only those roles that are navigable and have a positive lower multiplicity bound are selected as these are the navigations an instance has to define in order to be a valid instance.

Definition 25 (Entity Neighbourhood Conformance)**:** As with local conformance the neighbourhood conformance of Entities delegates to the clabject operation.

$$\epsilon.\texttt{neighbourhoodConforms}(\epsilon') := \epsilon.\texttt{neighbourhoodConforms}_{clabject}(\epsilon')$$

Definition 26 (Connection Neighbourhood Conformance)**:** The neighbourhood conformance of connections not only includes the navigations the connection itself participates in, but also the clabjects reachable through the roles of the connection.

$$\delta.\texttt{neighbourhoodConforms}(\delta_t) := \delta.\texttt{neighbourhoodConforms}_{clabject}(\delta_t) \wedge$$
$$\forall rN \in \delta_t.\texttt{roleNames}() :$$
$$\quad \delta.\texttt{navigate}(rN).\texttt{localConforms}(\delta_t.\texttt{navigate}(rN))$$

The existence and conformance of the roles at δ is already ensured by the local conformance of the connections, so it does not need to be checked again.

6.4 Multiplicity Conformance

A connection holds a multiplicity for every destination. As the multiplicity is a statement about the number of instance connections that can exist in the classified domain, it does not make sense to define multiplicities on the lowest ontological level.

The informal meaning of a multiplicity [1..2] for one participant A of a connection B is that every instance of any *other* participant of B must be connected to at least one and at most two instances of A through instances of B. One connection cannot conform or negate the multiplicity constraint, only the whole classified domain can. As a consequence, the check operation does not have a parameter.

Informally, the *instances* of both the connection and the participating instances are counted and checked against the multiplicity. At this point the set of the instances of a connection and/or clabject is not yet defined. Multiplicity conformance is a prerequisite for being considered an instance. So the set of clabjects that need to checked can be determined by neighbourhood conformance.

$$
\begin{aligned}
&\delta.\texttt{multiplicityConforms}() := \\
&\forall \psi \in \delta.\texttt{roles}() : \\
&\quad \forall \psi' \in \delta.\texttt{roles}() : \psi \neq \psi' : \\
&\quad\quad \forall \gamma \in \Sigma_{\delta.\mathsf{level}+1}.\texttt{clabjects}() : \\
&\quad\quad\quad \gamma.\texttt{neighbourhoodConforms}(\psi'.\mathsf{destination}) : \\
&\quad\quad\quad \psi.\mathsf{lower} \leq \\
&\quad\quad\quad\quad |\{\psi'' \in \Sigma_{\delta.\mathsf{level}+1}.\texttt{roles}() : (\psi''.\mathsf{destination} \neq \gamma) \wedge \\
&\quad\quad\quad\quad (\gamma \in \psi''.\mathsf{connection}.\texttt{participants}()) \\
&\quad\quad\quad\quad \wedge (\psi''.\mathsf{roleName} = \psi.\mathsf{roleName}) \wedge \\
&\quad\quad\quad\quad (\psi''.\mathsf{connection}.\texttt{neighbourhoodConforms}(\delta))\}| \\
&\quad\quad\quad\quad \leq \psi.\mathsf{upper}
\end{aligned}
$$

The main problem of the multiplicity check is that it is part of the isInstance check. Combined with the design requirement to avoid recursion at this level, the consequence is that the multiplicity check cannot rely on the instances. The set of clabjects that need to have a conforming number of navigations for the current multiplicity constraint is given by the clabjects γ that neighbourhood conform to the destination of the current role ψ' opposite the checked role ψ. The set of roles ψ'' matching the constraint cannot point to γ itself but γ has to be a part of the connection. So ψ'' defines a navigation for γ. For the navigation to be a valid one its connection has to neighbourhood conform (the strictest check possible) δ and the roleName has to be the same.

6.5 Property Conformance

Property conformance checks that a clabject has all the properties required to be an instance of a type. The step from property conformance to actual classification is small and sometimes trivial, but still necessary as property conformance is required for classification, but not sufficient, so there may be property conforming clabjects that are not instances. See 6.6.1 for details.

Definition 27 (Clabject Property Conformance)**:** Property conformance takes the type clabject as input and checks the candidate instance for the presence of a conforming property for every property of the type.

$$\begin{aligned}&\gamma_i.\texttt{propertyConforms}_{clabject}(\gamma_t) := \gamma_i.\texttt{neighbourhoodConforms}(\gamma_t) \wedge \\ &(\forall \psi_t \in \gamma_t.\texttt{navigations}() : \psi_t.\mathsf{lower} > 0 \wedge \psi_t.\mathsf{connection.potency} > 0 : \\ &\quad \exists \psi_i \in \gamma_i.\texttt{navigations}() : \\ &\quad\quad \psi_i.\mathsf{connection}.\texttt{propertyConforms}(\psi_t.\mathsf{connection})) \\ &\wedge \neg \gamma_i.\texttt{isInstanceOfExcluded}(\gamma_t)\end{aligned}$$

All Features are included in `neighbourhoodConformance` as they fit in the tighter scope. The extra element in property conformance is the recursive call to the property conformance of the connections connecting the clabjects (only those which must exist according to the classified domain). With the definition of connection property conformance, the check quickly spans all reachable clabjects. As a consequence, inside this closure either all clabjects are property conforming or none is.

The operation `isInstanceOfExcluded` ensures that there are no contradictions between expressed generalizations and classifications regarding the property conformance in question (see 6.5.1 for details). By weighting the expressed correlations above the ontological properties, they are primary information for the judgement of the classification question.

The definition of `propertyConformance` is recursive because it calls the same operation of the connections the subject takes part in. The resolution of this recursion is subject to section 6.7.

Definition 28 (Entity Property Conformance)**:** Property conformance for en-

tities follows the same rule as in the previous definitions in not adding anything to the clabject definition.

$\epsilon_i.\texttt{propertyConforms}(\epsilon_t) := \epsilon_i.\texttt{propertyConforms}_{clabject}(\epsilon_t)$

Definition 29 (Connection Property Conformance)**:** Clabject property conformance recursively branches to the connections. The clabject definition alone would include all the clabjects that participate in a connection (starting from the source). The addition to the reachable clabjects are the participants of the connection. Naturally, they are included in connection property conformance.

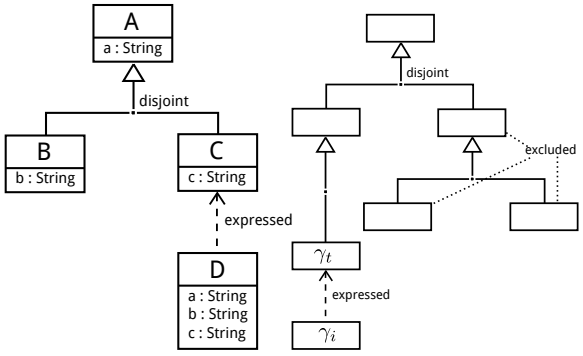
$\delta_i.\texttt{propertyConforms}(\delta_t) := \delta_i.\texttt{propertyConforms}_{clabject}(\delta_t) \wedge$
$\delta_t.\texttt{multiplicityConforms}() \wedge$
$\forall \psi_{\in} \delta_t.\texttt{roles}() :$
$\quad \exists \psi_i \in \delta_i.\texttt{roles}() :$
$\quad\quad \psi_i.\mathsf{roleName} = \psi_t.\mathsf{roleName} \wedge \psi_i.\mathsf{navigable} = \psi_t.\mathsf{navigable} \wedge$
$\quad\quad \psi_i.\mathsf{destination}.\texttt{propertyConforms}(\psi_t.\mathsf{destination})$

Besides clabject property and multiplicity conformance the connection check branches to the participating clabjects. Together with the propertyConformance check of these clabjects (regardless of whether they are entities or connections), the check branches to all the connections they participate in, thus spanning all clabjects reachable through connections from the source clabject.

6.5.1 Excluded Types through Generalization

The properties checked by the property conformance operation include one additional aspect beyond the properties (navigations and features). If the type takes part in a disjoint generalization, any clabject can only ever be an instance of one of the subtypes, but not both[1]. So if there is a classification stating that the subject is already an instance of a subtype, the other subtypes are excluded from the possible types of the subject. It is impor-

[1] By weighting the disjointness of the generalization over the properties, it is primary information



(a) Single Exclusion (b) Generic Domain

Figure 6.1: The disjoint sibling problem

tant to note that the instanceOf relationship indicated by the classification does not need to be reanalyzed based on properties. This information is not questioned in this operation, only processed. Given the model situation shown in figure 6.1a, there are two observations: First, `D` has all the properties to property conform to `B`. Second, `D` cannot be an instance of `B` as it is already an instance of `C` and the generalization states that the two are disjoint. So as a conclusion, there are correlations in the model prohibiting the classification of `D` to `B` but it is neither a feature nor a navigation. Figure 6.1b shows a more general picture. The exclusion is not limited to single types, but whole subtrees of the inheritance hierarchy can be excluded from property conformance by disjoint generalizations. The operation γ_i.`isInstanceOfExcluded`(γ_t) returns true if γ_i cannot be an instance of γ_t. Operation 12 gathers all the generalizations *classGener* on the type level that are disjoint. The set *possibles* holds the supertypes of the type, i.e. all clabjects that could possibly be excluded from the possible types of γ_i if there is a classification between γ_i and γ_t. The relevant classifications *insts* are gathered by searching for those with γ_i or one of its supertypes as their instance.

All the *classGener* are processed. If the subtypes are not disjoint from

Operation 12 γ_i.isInstanceOfExcluded(γ_t)

```
classGener ← {ξ ∈ Σ_{γt.level} : ξ.disjoint}
possibles ← {γt} ∪ γt.modelSupertypes()
excluded ← ∅
insts ← {φ ∈ Σ_{γi.level} :
  φ.expressed ∧ (φ.instance = γi ∨ φ.instance ∈ γi.modelSupertpyes())}
for ξ ∈ classGener do
  if possibles ∩ ξ.subtype ≠ ∅ then
    excluded ← excluded ∪ (ξ.subtype \ possibles)
for φ ∈ insts do
  actuals ← {φ.type} ∪ φ.type.modelSupertypes()
  if actuals ∩ excluded ≠ ∅ then
    return true
return false
```

the possibles it means there is a generalization that partitions a supertype of γ_t and another clabject and therefore excludes some part of the model from the possible types of γ_i, namely the remaining subtypes of the generalization which are not supertypes of γ_t (those cannot be excluded, as an instance of the subtype is always an instance of the supertype as well).

The remaining thing to check is whether there actually exists some classification between γ_i and one of the actually excluded types. If the type (or any of its supertypes) of the classification is one of *excluded*, the operation returns True meaning that γ_i is already an expressed (through a classification) instance of a disjoint sibling[1] of γ_t and therefore cannot property conform to γ_t.

[1] two clabjects are siblings if they have a common supertype expressed by generalization(s).

6.6 Classification

After property conformance, all the necessary ingredients to define classification are in place. However, classification cannot be defined right away as classification is an umbrella for two disjoint types of classification that need to be established first.

6.6.1 Property Conformance and Classification

The delta from property conformance to classification is twofold. Property conformance is a strict requirement of classification. There never is an instance that is not property conforming.

There are two kinds of classification, hyponymic classification and isonymic classification. Hyponyms and isonyms completely partition the instances of a type, so there is no instance of any type that is not either a hyponym or an isonym.

6.6.2 Additional Property Definition

The distinction between isonyms and hyponyms is based on whether or not the instance extends the set required by the type or not. To judge the extension of the type definition, the operation `hasAdditionalProperties` checks if the candidate instance defines any properties that are not required to be an instance of the type.

$$
\begin{aligned}
&\gamma_i.\texttt{hasAdditionalProperties}(\gamma_t) := (\exists \eta_i \in \gamma_i.\texttt{features}() : \\
&\quad \nexists \eta_t \in \gamma_t.\texttt{features}() : \eta_i.\texttt{conforms}(\eta_t) \\
&\quad) \vee (\\
&\quad \exists \psi_i \in \gamma_i.\texttt{navigations}() : \nexists \psi_t \in \gamma_t.\texttt{navigations}() : \\
&\qquad \psi_i.\mathsf{roleName} = \psi_t.\mathsf{roleName})
\end{aligned}
$$

6.6.3 Isonyms and Hyponyms

Definition 30 (Isonym)**:** Informally a clabject is an isonym of a type if it could have been created from that type. An isonym is an instance that has all the properties required of its type, but no more. "Isonym" is a relative term to the type the instance is an isonym of (the instance is called an isonymic instance of the type) If an isonym has been created by instantiating a type, it is called an offspring of that type, and the type is called the blueprint of the instance. The type of an isonym is called a complete type of the clabject since it contains a description of all the properties of the clabject. Besides property conformance, the formal definition requires potency conformance as well as not adding any properties to those required by the type.

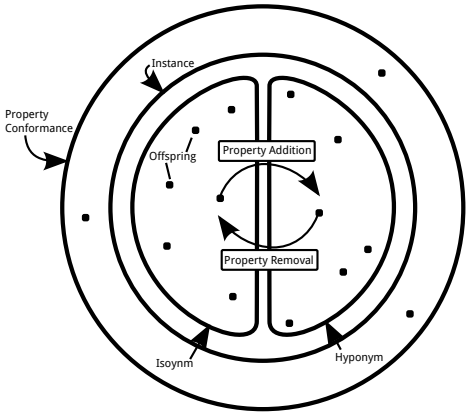
$$\gamma_i.\texttt{isIsonym}(\gamma_t) := \gamma_i.\texttt{propertyConforms}(\gamma_t) \wedge \\ \neg\gamma_i.\texttt{hasAdditionalProperties}(\gamma_t) \wedge \\ (\gamma_t.\mathsf{potency} \neq * \implies \gamma_t.\mathsf{potency} = \gamma_i.\mathsf{potency} + 1)$$

Definition 31 (Hyponym)**:** A hyponym is the other kind of instance a clabject can be. A clabject is a hyponym of a type if it is an instance of the type but could not have been created from it because it does not define all of the clabject's properties. A hyponym, or a hyponymic instance of a type is an instance that defines more properties than needed by the definition of the type. The type of a hyponym is called an incomplete type because the type does not contain enough properties to produce the instance.

$$\gamma_i.\texttt{isHyponym}(\gamma_t) := \gamma_i.\texttt{propertyConforms}(\gamma_t) \wedge \\ \gamma_i.\texttt{hasAdditionalProperties}(\gamma_t)$$

Definition 32 (IsInstance)**:** "Instance" is the umbrella term used to describe either isonyms or hyponyms. With the help of the *isInstance* operation, the two can now be described in relation to one another. A hyponym is an instance that is not an isonym. Every instance is either an isonym or a hyponym. So the set of isonyms and the set of hyponyms completely and disjointedly partition the set of instances of a given type. Isonym is

Figure 6.2: Instances and Property Conformance Partition



the complement of hyponym in the set of instances and hyponym is the complement of isonym in the set of instances of a type. If an isonym adds a Feature to its definition it becomes a hyponym and any hyponym is able to become an isonym by stripping down its properties to the minimal ones needed to stay an instance of the type. There is no other way to check whether a clabject is an instance of a type other than checking whether it is an isonym or a hyponym.

$$\gamma_i.\texttt{isInstance}(\gamma_t) := \gamma_i.\texttt{isIsonym}(\gamma_t) \vee \gamma_i.\texttt{isHyponym}(\gamma_t)$$

However the checks can be broken down to their canonical components and therefore provide a pseudo independent definition.

$$\gamma_i.\texttt{isInstance}(\gamma_t) := \gamma_i.\texttt{propertyConforms}(\gamma_t) \wedge$$
$$(\neg\gamma_i.\texttt{hasAdditionalProperties}(\gamma_t) \implies$$
$$(\gamma_t.\mathsf{potency} \neq * \implies \gamma_t.\mathsf{potency} = \gamma_i.\mathsf{potency} + 1))$$

6.6.4 Property conforming non-instances

As the definitions show, a clabject can be property conforming and still not be an instance. If the clabject defines additional properties, it is a hyponym and thus an instance. If it does not define any additional properties, it can be an isonym (not a hyponym) but only if its potency is

correct. So a property conforming clabject that does not define any additional properties and has the wrong potency is not an instance of the type.

$$(\gamma_i.\texttt{propertyConforms}(\gamma_t) \wedge \neg\gamma_i.\texttt{hasAdditionalProperties}(\gamma_t) \wedge$$
$$\neg(\gamma_t.\mathsf{potency} \neq * \implies \gamma_t.\mathsf{potency} = \gamma_i.\mathsf{potency} + 1)) \implies$$
$$\neg\gamma_i.\texttt{isInstance}(\gamma_t)$$

When can such quasi-instances occur? The most likely case is when the modelling mode was switched from constructive to exploratory and the potency does not (yet) match. The potency will probably be higher than the depth of the existing isonym tree because the isonyms are simply not yet existent. So the occurrence of a property conforming quasi-instance is most likely a user error. Regardless of their origin, property conforming quasi-instances are not desirable and should be corrected immediately when they appear.

6.6.5 The value of hyponyms

Although it might seem that hyponyms are much less important than isonyms, this is not the case. Especially in constructive modeling, they are the basis for the polymorphic assignment and dynamic binding capabilities that underpin object-oriented programming. The typing rules of object-oriented programming languages like Java basically ensure that variables of a given type can have hyponyms as well as isonyms of that type assigned to them. Consider the following Java statement:

Ant z = **new** MaleAnt();

z is an isonym of MaleAnt and a hyponym of Ant. Ant is specified as the static type, so any expression on z can only make use of the Ant-part of the interface. As MaleAnt is a subtype of Ant, any instance can play the role of an Ant according to the Liskov substitution principle (48). The distinction between isonyms and hyponyms provides the formal foundation for this mechanism.

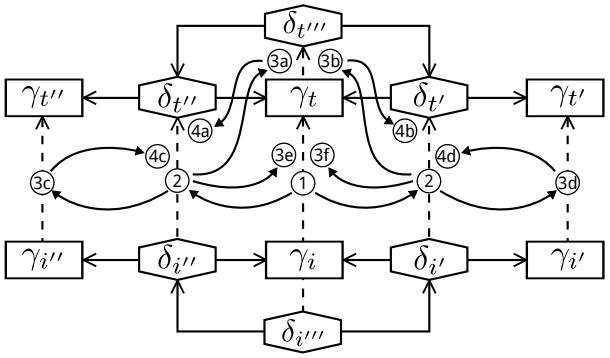


Figure 6.3: Recursion and marking of γ_i.isInstance(γ_t)

6.7 Recursion Resolution

Property conformance requires the recursive analysis of all connected (and connecting) clabjects. Without a terminating condition, the execution will enter an infinite loop, as the analysis process will be immediately passed back to the calling clabject. The terminating condition is simple: If a pair of clabjects is analyzed for a second time it means that the analysis process has not yet terminated to false since no information has been found negating the claim. There will thus be no discovery of negating information in a second analysis so true is returned instead.

Upon the first analysis of any pair of clabjects γ_i.propertyConforms(γ_t), the pair can be marked. If the pair is then visited a second time, the mark is detected and returned.

Figure 6.3 shows the recursion and marking strategy for the computation of the operation γ_i.propertyConforms(γ_t). First

γ_i.neighbourhoodConforms(γ_t)

is checked (1). γ_i.isInstanceOfExcluded(γ_t) does not hold as there are no generalizations. The check spans to the connections

$\delta_{\{i',i''\}}$.propertyConforms($\delta_{\{t',t''\}}$)(2).

After $\delta_{\{t',t''\}}$.multiplicityConformance() (not shown in figure 6.3) and

neighbourhood conformance the check branches to the connections that $\delta_{\{t',t''\}}$ take part in as participants, leading to

$\delta_{i'''}$.propertyConforms($\delta_{t'''}$)(3a,3b).

As $\delta_{t'''}$ does not take part in any connections the next check is the classification of the participants (4a,4b). Since this is the special case where the participants are the last analysed clabjects, these two branches terminate to true as they have reached already analysed clabjects ($\delta_{\{t',t''\}}$) and did not find contradicting information along the way. (3c-f) originates from (2) as well. (3e,f) terminate immediately as well as the analysis of the participant that spawned (2). (3c,d) have no further connections in this example than $\delta_{\{t',t''\}}$ so the checks spawning from (3d,c) terminate as well. The overall check is then true given that the hidden parts of the example (features and traits) conform.

Chapter 7

Ontology Query Services

Now that all the ingredients are ready, the services that are visible to the end user when modeling or using an ontology can be defined. The services presented in this chapter comprise all the services that provide information about the ontology without changing its state in any way. These questions can range from logical properties to retrieving parts of a model based on certain criteria. A special kind of query is a boolean query checking for a certain property. A service qualifies as an ontology query if it does not alter any data inside the ontology, in other words: if it is side effect free.

7.1 Well Formedness Constraints

Well Formedness constraints are invariants which have to hold at all time otherwise an ontology is malformed. The well formedness of the whole ontology (or a model) recursively includes the well formedness of the contained elements, so inside a well formed model there are no malformed elements.

Definition 33 (Ontology well formedness)**:** Ontology elements can neither be computed, nor irrelevant. The concepts only apply to finer grained portions of the domain. Ontologies and models are both containers for the other elements and their well formedness depends on the well formedness of their

contained elements.

$\chi.\texttt{isWellFormed}() =: \chi.\mathsf{expressed} \wedge \chi.\mathsf{relevant} \wedge$
$\forall \Sigma : \Sigma.\texttt{isWellFormed}()$

Definition 34 (Model well formedness)**:** Models can only hold clabjects of the same level, the level that defines the model. Although correlations do not have a level, they connect clabjects, which in turn define their level and this level also has to match the model level.

$\Sigma_i.\texttt{isWellFormed}() =: (\forall \gamma \in \Sigma_i.\mathsf{children} \cap \mathsf{Clabject} :$
$(\gamma.\mathsf{level} = i) \wedge \texttt{clabjectValidSupertypePotency}(\gamma)) \wedge$
$(\forall \lambda \in \Sigma_i.\mathsf{children} \cap \mathsf{Correlation} : \lambda.\mathsf{level} = i) \wedge$
$\texttt{noCircularGeneralizations}(\Sigma_i) \wedge$
$(\forall \alpha \in \Sigma_i.\mathsf{children} : \alpha.\texttt{isWellFormed}())$

The potencies of the clabjects have a well-formedness constraint depending not only on the clabject itself, but on its supertypes. If a clabject does not add any new Properties to the definition of its supertype, any isonym of the supertype will also be an isonym of the subtype[1]. As a consequence the isonym trees will be the same and so the potencies of the clabjects always have to be identical. The operation $\texttt{clabjectValidSupertypePotency}(\gamma)$ ensures this for γ.

$\texttt{clabjectValidSupertypePotency}(\gamma) :=$
$(\{\eta \in \gamma.\texttt{eigenFeatures}() : \eta.\mathsf{durability} > 0\} = \emptyset$
$) \wedge ($
$\{\psi \in \gamma.\texttt{navigations}() \cup \gamma.\texttt{modelNavigations}() :$
$\psi.\mathsf{connection}.\mathsf{potency} > 0\} = \emptyset$
$) \implies (\forall \gamma_s \in \{\xi.\text{supertype} \forall \xi : \gamma \in \xi.\text{subtype}\} :$
$\gamma.\text{potency} = \gamma_s.\text{potency})$

The constraint `clabjectValidSupertypePotency` may not be satisfiable by some model constellations. If a subtype does not add any new properties to a supertype, it does not add any new properties to any of its direct supertypes. So if γ has more than one direct supertype (i.e. there is more than one generalization with γ as the subtype) and the supertypes have different

[1] The details are covered in 6

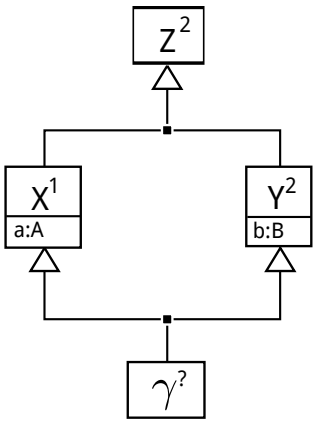


Figure 7.1: Unsolvable potency for γ

potencies, the requirement is not satisfiable by γ as it must have two different potency values. The situation is shown in figure 7.1. While theoretically possible, the constellation is not very likely to happen in practice as shallow subtypes (which do not add any properties) are usually used to partition a well-defined generalization or to introduce a new name. A shallow subtype of two types with different potency would form the union of two concepts with a different domain lifespan. However if this situation occurs it has to be resolved by the user.

Another constraint is that there must not be circular generalizations in the sense that any clabject can reach itself by navigating only subtype (or supertype) ends of generalizations. The operation

$\texttt{noCircularGeneralizations}(\Sigma)$

ensures this for the model Σ.

$$\texttt{noCircularGeneralizations}(\Sigma) := \forall\gamma \in \Sigma.\texttt{clabjects}() : \gamma \notin \gamma.\texttt{modelSupertypes}()$$

Definition 35 (Clabject well formedness)**:** A clabject is well-formed if potency and level are not negative (* values always conform). Each clabject is a namespace for Features, meaning that within the children of a clabject the name of a Feature is unique.

$\gamma.\texttt{isWellFormed}() := (\gamma.\mathsf{potency} \geq 0) \wedge$
$(\gamma.\mathsf{level} \geq 0) \wedge$
$\forall \eta \in \gamma.\texttt{eigenFeatures}() : \nexists \eta' \in \gamma.\texttt{eigenFeatures}() : \eta.\mathsf{name} = \eta'.\mathsf{name} \wedge$
$\eta \neq \eta'$

For a Feature, the durability must not be negative.

$\eta.\texttt{isWellFormed}() := (\eta.\mathsf{durability} \geq 0)$

An attributes mutability cannot be higher than its durability.

$\zeta.\texttt{isWellFormed}() := \zeta.\mathsf{mutability} \geq 0 \wedge \zeta.\mathsf{mutability} \leq \zeta.\mathsf{durability}$

Definition 36 (Connection well formedness)**:** Inside a connection, the roleNames have to be unique as they are the key for accessing the participants of a connection. A connection cannot connect clabjects across multiple levels, i.e. the participants of a connection all have to be on the same level. Also, the potencies of the participating clabjects cannot be lower than the potency of the connection. A connection does not necessarily need to have roles, it can also inherit them from supertypes. As consequence, if a connection does not have any roles, it has to inherit them. So it is only valid for a connection to have no roles if there is an outgoing generalization.

$\delta.\texttt{isWellFormed}() := (|\{\psi.\mathsf{roleName} : \psi \in \delta.\texttt{roles}()\}| = |\delta.\texttt{roles}()|) \wedge$
$(|\{\psi.\mathsf{destination.level} : \psi \in \delta.\texttt{roles}()\}| = 1) \wedge$
$(\forall \gamma \in \delta.\texttt{participants}() : \gamma.\mathsf{potency} \geq \delta.\mathsf{potency}) \wedge$
$|\delta.\texttt{eigenRoles}()| = 0 \implies \exists \xi : \delta \in \xi.\mathsf{subtype} \wedge$
$\nexists \xi : \delta \in \xi.\mathsf{subtype} \implies |\delta.\texttt{eigenRoles}()| \geq 2$

Definition 37 (Role well formedness)**:** For roles, the multiplicities have to conform, which means no negative values and the lower bound is never higher than the upper. Either both multiplicities are present or neither. If they are not present, the potency of the defining connection has to be zero.

$\psi.\texttt{isWellFormed}() := (\neg\psi.\mathsf{lower} \iff \neg\psi.\mathsf{upper}) \wedge$
$(\psi.\mathsf{connection.potency} = 0 \iff \neg\psi.\mathsf{lower} \wedge \neg\psi.\mathsf{upper}) \wedge$
$(\psi.\mathsf{lower} \geq 0 \wedge \psi.\mathsf{upper} \geq \psi.\mathsf{lower})$

Definition 38 (Correlation well formedness)**:** Generalizations and SetRelationships can only connect clabjects at the same level. Generalizations are not valid between potency zero clabjects. Classifications must connect two adjacent levels. There can be no inheritance between entities and connections.

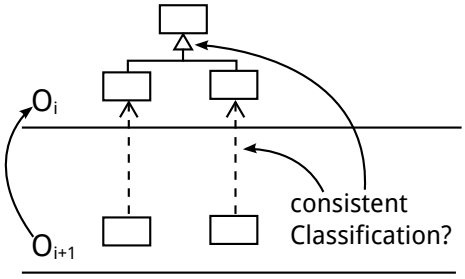
$$\begin{aligned}
&\xi.\texttt{isWellFormed}() := |\gamma.\mathsf{level} : \gamma \in \{\xi.\mathsf{subtype} \cup \xi.\mathsf{supertype}\}| = 1 \wedge \\
&\quad (\xi.\mathsf{disjoint} \vee \xi.\mathsf{complete}) \implies |\xi.\mathsf{subtype}| \geq 2 \wedge \\
&\quad \xi.\mathsf{intersection} \implies |\xi.\mathsf{supertype} \geq 2| \wedge \\
&\quad \forall \gamma \in \{\xi.\mathsf{subtype} \cup \xi.\mathsf{supertype}\} : \gamma.\mathsf{potency} > 0 \wedge \\
&\quad |\{\gamma.\texttt{linguisticType}() : \gamma \in (\xi.\mathsf{subtype} \cup \xi.\mathsf{supertype})\}| = 1 \\
&\phi.\texttt{isWellFormed}() := (\phi.\mathsf{instance.level} + 1 = \phi.\mathsf{type.level}) \\
&\upsilon_i.\texttt{isWellFormed}() := (\upsilon_i.\mathsf{base.level} = \upsilon_i.\mathsf{inverse.level}) \wedge \\
&\quad (\{\upsilon_i.\mathsf{base}, \upsilon_i.\mathsf{inverse}\} \subset \mathsf{Connection} \wedge \upsilon_i.\mathsf{base}.\texttt{order}() = 2 \wedge \\
&\qquad \upsilon_i.\mathsf{inverse}.\texttt{order}() = 2) \\
&\upsilon_e.\texttt{isWellFormed}() := \upsilon_e.\mathsf{base.level} = \upsilon_e.\mathsf{equal.level} \\
&\upsilon_c.\texttt{isWellFormed}() := \\
&\quad \upsilon_c.\mathsf{base.level} = \upsilon_c.\mathsf{complement.level} \wedge \upsilon_c.\mathsf{base.level} = \upsilon_c.\mathsf{universe.level}
\end{aligned}$$

7.2 Consistency, Completeness & Validity

With the tools to identify classification through artefacts at hand the path is clear to define the validity of an ontology. An ontology is valid if it does not contain any contradictory information and is complete from the perspective of the modeling mode used.

Contradictory situations occur when the logical meaning of two parts of an ontology are incompatible. A simple example is an ontology containing an equals correlations between two clabjects which are clearly not equal. Validity can also be violated by implicit claims about the presence of artefacts which are not in fact present in the ontology. Such information incompatibilities usually spans multiple levels, as the claims are statement on the instances of the types such as a disjoint generalization claiming that no instance of the supertype is an instance of more than one subtype. From

Figure 7.2: Consistent Classification



the type level alone, a claim cannot be refuted as it needs a classified models to either satisfy or violate the claim.

7.2.1 Consistent Classification

Adjacent pairs of models are consistent with one another if the logical relationships between them are consistent. Classifications and generalizations are logical relationships between models. Classifications directly link an instance to a type and generalizations impose constraints on the instances of the involved types with their boolean traits.

Definition 39 (Classification Consistency)**:** A classification is consistent if the instance actually is an instance of the type and the classification correctly characterizes the nature of their relationship.

ϕ.`isConsistent`() :=
ϕ.instance.`isInstance`(ϕ.type)$\wedge$
ϕ.kind = *hyponym* $\iff$ ϕ.instance.`isHyponym`(ϕ.type)$\wedge$
ϕ.kind $\in \{isonym, instantiation\}$ $\iff$ ϕ.instance.`isIsonym`(ϕ.type)

In terms of consistency, *instantiation* and *isonym* are the same. It is not possible to judge retrospectively whether an instance has been created from a type as long as the information shows that it could have been created from the type. A classification of kind *instance* does not make any claim about the precise nature of the classification (i.e. whether it is isonymic or hyponymic), juts that it is one or other of them.

Definition 40 (Generalization Consistency)**:** A generalization as such cannot be wrong as it states that the subtypes inherit the properties of the supertypes. A generalization introduces new information, whereas a classification makes already existing implicit information explicit. The boolean traits of a generalization in contrast can be checked. For example, a disjoint generalization must partition the set of instances, an overlapping (not disjoint) generalization must have intersecting sets of instances and an intersection must be the union of the supertype instances.

$$
\begin{aligned}
&\xi.\texttt{isConsistent}() := \\
&(\xi.\mathsf{disjoint} \iff \forall \gamma_i \in \Sigma_{\xi.\mathsf{supertype.level}+1}.\mathsf{clabjects} : \\
&\quad \gamma_i.\texttt{isIsonym}(\xi.\mathsf{supertype}) : \\
&\quad\quad |\{\phi \in \Sigma_{\xi.\mathsf{supertype.level}+1}.\mathsf{classifications} : \\
&\quad\quad\quad \gamma_i = \phi.\mathsf{instance} \wedge \phi.\mathsf{type} \in \xi.\mathsf{subtype}\}| \leq 1 \\
&) \wedge (\\
&\xi.\mathsf{complete} \iff \forall \gamma_i \in \Sigma_{\xi.\mathsf{supertype.level}+1}.\mathsf{clabjects} : \\
&\quad \gamma_i.\texttt{isIsonym}(\xi.\mathsf{supertype}) : |\{\gamma_t \in \xi.\mathsf{subtype} : \gamma_i.\texttt{isIsonym}(\gamma_t)\}| \geq 1 \\
&) \wedge (\\
&\xi.\mathsf{intersection} \iff \forall \gamma_i \in \Sigma_{\xi.\mathsf{supertype.level}+1}.\mathsf{clabjects} : \\
&\quad \forall \gamma_t^s \in \xi.\mathsf{supertype} : \gamma_i.\texttt{isIsonym}(\gamma_t^s) : \\
&\quad \gamma_i.\texttt{isIsonym}(\xi.\mathsf{subtype}))
\end{aligned}
$$

A generalization is either disjoint or overlapping (i.e. not disjoint) and cannot be consistent with a constellation of instances if they do not adhere to the stated fact. A special case could be when there are no instances at all, but in this case it becomes clear that then the generalization is actually disjoint and complete. To be overlapping there needs to be an instance which is an instance of more than one subtype. If there are no instances, such a situation does not exists. This is not the case with disjointness - to be disjoint every instance must be an instance of at most one subtype. For completeness the case is similar. To be incomplete, there must exists an instance that is not an instance of any of the subtypes. If there are no instances, such a situation does not exist. To be complete such an instance must not exist, and if there are no instances, the constraint holds. For intersections the case is the same. If a generalization is not an intersection

there has to be an instance to negate the fact that an instance of all the supertypes is also an instance of the subtype, meaning there has to be an instance of all the supertypes that is not an instance of the subtype.

Disjoint generalizations play a special role because their information is already used in the check for property conformance. If a clabject is an expressed instance of a subtype in a disjoint generalization it cannot be an instance of another ones of the subtypes, even if it satisfies all other domain constraints. If the classification is expressed it is assumed to be user input and thus given higher weight than other constraints. If the consistency check for disjoint generalizations used the isIsonym operation like the other checks, the automatically triggered propertyConformance check would weigh the disjoint generalization higher than the artefacts that are subject to the check. In the end, a disjoint generalization would always be consistent as a clabject can only ever be an isonym of one of the subtypes according to the definition. Hence, expressed classification is the fall-back to determine the instances to be checked.

Definition 41 (Consistent Model Classification)**:** The consistent classification of a model also involves the classifying model (see figure 7.2). If the subject model does not have a classifying model, it cannot be consistently classified as it is not classified at all, meaning that its clabjects do not have any types. A prerequisite for consistent classification (in fact for any sensible reasoning) is the well formedness of the subject model as well as its classifying model. Generalizations make statements about instances, so the consistency of the generalizations in the classifying model is ensured by the clabjects of the subject model. The logical relations between the subject model and its classifying model comprise the generalizations of the classifying model and the classifications of the subject model.

Σ_i.`isConsistentlyClassified`() :=
$\exists \Sigma_{i-1} \wedge \Sigma_i$.`isWellFormed`() $\wedge$ Σ_{i-1}.`isWellFormed`()$\wedge$
$\forall \phi \in \Sigma_i$.classifications : ($\phi$.expressed $\implies$ ϕ.`isConsistent`())$\wedge$
($\forall \xi \in \Sigma_{i-1}$.generalization : ξ.`isConsistent`())

Definition 42 (Ontology Consistency)**:** The consistency of an ontology requires that all the information inside the ontology is valid, but the ontology itself does not need to be complete. A piece of information is valid if it does not conflict with any other statement in the ontology. Formally, consistency is equivalent to the consistent classification of all the models except the root model.

$$\chi.\texttt{isConsistent}() := \forall \Sigma_i, i > 0 : \Sigma_i.\texttt{isConsistentlyClassified}()$$

Definition 43 (Constructive Ontology Validity)**:** The only claim an ontology has to satisfy in constructive mode in order to be valid is that it does not contain any logical conflicts.

$$\chi.\texttt{isValid}_{constructive}() := \chi.\texttt{isConsistent}()$$

Definition 44 (Potency Completeness)**:** The potency of a clabject represents the actual depth of the isonym tree. So if a clabject has potency > 1, it is not sufficient that there exists an isonym, but at least one of these isonyms has to have an isonym as well. Classifications do not make a statement about potency, they just indicate relations between two adjacent levels, hence they can only check potencies of value 1.

$$\begin{aligned}
&\gamma.\texttt{isPotencyComplete}() := \\
&(\gamma.\mathsf{potency} = 0 \implies \nexists \gamma_i \in \Sigma_{\gamma.\mathsf{level}+1}.\mathsf{clabjects} : \\
&\quad (\gamma_i.\texttt{propertyConforms}(\gamma) \wedge \neg \gamma_i.\texttt{hasAdditionalProperties}(\gamma))) \wedge \\
&((\gamma.\mathsf{potency} \neq * \wedge \gamma.\mathsf{potency} \neq 0) \implies \\
&\quad (\exists \gamma_i \in \Sigma_{\gamma.\mathsf{level}+1}.\mathsf{clabjects} : \gamma_i.\texttt{isIsonym}(\gamma)) \wedge \\
&\quad (\forall \gamma_i \in \Sigma_{\gamma.\mathsf{level}+1}.\mathsf{clabjects} : \gamma_i.\texttt{isIsonym}(\gamma) : \gamma_i.\texttt{isPotencyComplete}()))
\end{aligned}$$

If a clabject has potency 0, it cannot have an isonym. The statement is trivial as the check for isonyms requires the potency of the isonym to be one lower and 0 is the lowest possible value for potency. So the check for isonyms can never be true on a potency 0 clabject. The potency completeness goes further. If a clabject has potency 0, there must not be an element that would be an isonym if the potency were 1. The check for isonyms has three parts: property conformance, no additional property definition and potency conformance. Denying the isonym property based on potency conformance

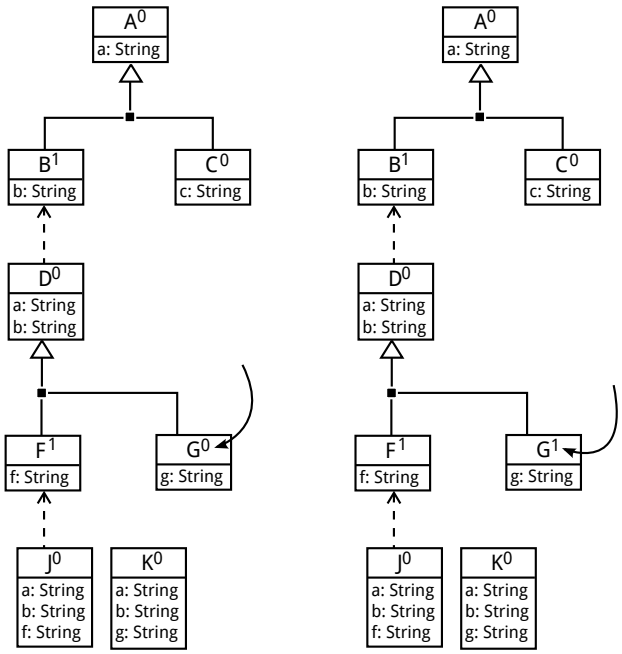


Figure 7.3: Changing of a potency value due to artefacts

is valid, as the isonym property is a statement on the current information in the ontology, but for potency completeness the potency value has to reflect the actual situation. In other words if the potency of a clabject should be 1 because there exists an isonym, the potency cannot be 0.

Definition 45 (Ontology Completeness)**:** An ontology is complete if it is not only consistent but also complete in the sense that all statements about the ontology are true based on the information currently in it. Formally the delta to consistency is the potency completeness of all the clabjects.

$$\chi.\texttt{isComplete}() := \chi.\texttt{isConsistent}() \wedge$$
$$\forall \Sigma_i : \forall \gamma \in \Sigma_i.\mathsf{clabjects} : \gamma.\texttt{isPotencyComplete}()$$

Definition 46 (Exploratory Ontology Validity)**:** In the context of ontology validity the difference between constructive and exploratory modeling mode is the assumed completeness of the model information. While in constructive

mode the absence of information cannot be an error (only the presence of wrong information can be), in exploratory mode the information is assumed to be complete. So if one statement relies on the presence of other model elements, exploratory correctness is only satisfied if the element actually exists (in constructive correctness it is enough if the element can exist in the future).

$\chi.\texttt{isValid}_{exploratory}() := \chi.\texttt{isComplete}()$

7.3 Inheritance

This section focuses on the detection of generalizations based on artefacts. In the default constructive use case of modeling there is no need to detect inheritance based on properties as inheritance is used to define those properties in the subtypes. So detecting generalizations (or subsumption) is an operation tailored towards the exploratory use case of detecting correlations based on artefacts.

Inheritance only occurs between clabjects and takes place on one ontological level only. The independence of inheritance from classified or classifying models makes the formal definition a lot easier. The basic principle of subtyping is borrowed from set theory. Every instance of a subtype is also an instance of the supertype. The set of instances of the subtype is a subset of the instances of the supertype. Processing the actual instances is not necessary (and also not guaranteed to work as there might not yet be any) because the type facade of types defines what instances will look like. The building blocks of the type facade are features and navigations. For features there are equality operations. As navigations depend on both the realizing connection and the target clabject, they are processed inside the main algorithm.

Definition 47 (Attribute equality)**:** For attributes to be equal, every directly stored piece of information has to be equal. This includes name, durability,

datatype and mutability. The value has to be equal if the mutability is zero, because any conforming feature has to have exactly that value. The ontological level or anything outside the scope of the attribute is not part of the definition. This means that attributes from different ontological levels can potentially be equal.

$\zeta.\texttt{equals}(\zeta') := \zeta.\mathsf{name} = \zeta'.\mathsf{name} \wedge$
$\zeta.\mathsf{durability} = \zeta'.\mathsf{durability} \wedge$
$\zeta.\mathsf{mutability} = \zeta'.\mathsf{mutability} \wedge$
$\zeta.\mathsf{datatype} = \zeta'.\mathsf{datatype} \wedge$
$\zeta.\mathsf{mutability} = 0 \implies \zeta.\mathsf{value} = \zeta'.\mathsf{value}$

Definition 48 (Method equality)**:** Name and durability are the same as for attributes. The input and output definitions have to be equal, and their equality in turn depends on the equality of the contained parameters. For primitive data, the equality computation is trivial while for PLM elements the definitions defined in this section apply. As with the other definitions, the method body is not part of the equality definition. Static checks cannot incorporate runtime semantics. All checks regarding methods are limited to the statically provided data.

$\eta.\texttt{equals}(\eta') := \eta.\mathsf{name} = \eta'.\mathsf{name} \wedge$
$\eta.\mathsf{input} = \eta'.\mathsf{input} \wedge$
$\eta.\mathsf{output} = \eta'.\mathsf{output}$

Durability zero features Since generalization is essentially a statement about the instances of the types, durability zero features do not matter because the instances will not be affected by them. The same holds for roles of potency zero clabjects. If they are neglected, equality does not hold for mutual subtypes, only similarity. So the following statement is only universally true if the durability zero facade is included:

$$\gamma.\texttt{isSubtype}(\gamma') \wedge \gamma'.\texttt{isSubtype}(\gamma) \iff \gamma.\texttt{equals}(\gamma').$$

Formally, it is more effort to filter the properties. In the following the definitions assume durability zero properties shall be neglected, as it is the

more complex case. A tool may provide an option to adjust the behaviour.

Subsumption does not need to be built incrementally on the model elements as in classification. For features, the requirement is already defined. Features have to be equal. Potency is not a part of subtyping. The facades of the sub- and supertype still include the inherited properties, but in many use cases there will be no correlation at all in the ontology. Even within the level, the conformance dependency remains intact. Subtyping is based on properties and contains recursion, just like classification.

$$\gamma.\texttt{subsume}(\gamma_s) := \gamma.\mathsf{level} = \gamma_s.\mathsf{level} \wedge$$
$$(\forall \eta_s \in \gamma_s.\texttt{features}() : \eta_s.\mathsf{durability} > 0 : \exists \eta \in \gamma : \eta.\texttt{equals}(\eta_s)) \wedge$$
$$(\forall \psi_s \in \gamma'.\texttt{navigations} : \psi_s.\mathsf{connection.durability} > 0 :$$
$$\exists \psi \in \gamma.\texttt{navigations} : \psi.\texttt{subtypeConforms}(\psi_s))$$

The subtype conformance of roles is defined separately as it is reused in the definition for connections again. A role subtype conforms to another role if it has the same roleName and navigability. The multiplicity of the conforming role may be stricter than the conformed to one.

$$\psi.\texttt{subtypeConforms}(\psi_s) := \psi_s.\mathsf{roleName} = \psi.\mathsf{roleName} \wedge$$
$$\psi_s.\mathsf{lower} \leq \psi.\mathsf{lower} \wedge \psi_s.\mathsf{upper} \geq \psi.\mathsf{upper} \wedge \psi_s.\mathsf{navigable} = \psi.\mathsf{navigable} \wedge$$
$$\psi.\mathsf{destination}.\texttt{subsume}() \wedge \psi.\mathsf{connection}.\texttt{subsume}(\psi_s.\mathsf{connection})$$

The addition for connections comes from the participants. The definition above covers everything that the clabject is connected to, while the addition covers what it connects.

$$\delta.\texttt{subsume}(\delta_s) := \gamma.\texttt{subsume}(\gamma_s) \wedge$$
$$|\delta.\texttt{roles}()| = |\delta_s.\texttt{roles}()| \wedge \forall \psi_s \in \delta_s.\texttt{roles}() :$$
$$\exists \psi \in \delta.\texttt{roles}() : \psi.\texttt{subtypeConforms}(\psi_s)$$

The inherent recursion can be solved with a marking mechanism similar to the one applied in classification. If a pair is visited for the second time, nothing is performed but true is returned instead. The implementing environment has to administer the checking and resetting of the marks. See 6.7

for details.

7.3.1 Shallow Subtyping

A subtype is a shallow subtype if it does not add anything to the type facade of the subtype. In other words, the clabjects produce the same isonyms. Formally, they are similar.

$\gamma.\texttt{isShallowSubtype}(\gamma_s) := \gamma.\texttt{similar}(\gamma_s)$

Referring to well formedness (7.1), shallow subtypes must have the same potency as their supertypes.

7.4 Ontology Validation Services

Ontology validations attempt to determine whether a whole ontology, or a part of an ontology, satisfies a certain property and the reason for the result can be explained by the engine to the user. The validity of the total ontology or a subset of it is very important to the user both during and after its creation. The selection of the part of the ontology to validate and the criteria used depends on the use case in hand:

Well formedness At any given time the well formedness of the ontology is very important. Well formedness is defined for every single element type and its definition includes recursion to the contained elements where necessary. As the check for one element is of constant complexity and the inclusion of all elements queries every element once at most, the well formedness can be called frequently. A tool might check every element every time its contents are changed and the whole ontology when saving or calling a bigger reasoning service.

Consistency The consistency of an ontology is a prerequisite for ontology validity and thus for most of the other services. As the consistency check spans the whole ontology and contains much redundancy if not

optimized by the implementation, the consistency should not be called too frequently, but certainly when saving the ontology state and when part of a bigger service query.

Completeness Completeness builds on consistency and is therefore fairly complex as well. Furthermore, completeness is of interest in constructive mode, so the trigger to check for ontology completeness upon saving can be derived either from the mode the ontology was last in or from a user preference.

Single element consistency As the ontology consistency check affects the whole ontology, but the consistency not necessarily of a single element, an engine could offer the user a shorthand way to check any correlation or provide an option to check any element when it, or any element connected to it, is changed. For example a classification could be checked for consistency if either the type or the instance is changed, as this change may very well affect the classification relationship.

7.5 Ontology Queries

Ontology queries allow users to ask for information about the ontology, such as selecting a portion of the ontology based on certain criteria or providing more detailed information about a single model element. Each user request is based on a question the user seeks to answer. These questions can range from correctness (validation services) to the retrieval of arbitrary model information but also questions tailored towards future states of the model such as the consequences of an operation in terms of a certain property. For example,

- Will an ontology still be correct if I perform an operation such as renaming, deletion, addition or the changing of a value.

- Will a pair of clabjects/a single clabject gain or loose a certain property if I perform such an operation?

7.5.1 Clabject Introspection Service

The reasoning engine can provide the following information about individual clabjects.

7.5.1.1 Types

The engine can compute the set of types of the instances in question, and for each type, can provide additional information such as:

modeled or computed If the type is modeled, there is an expressed classification element connecting the type and clabject. An optional operation is to check the consistency of this classification. If the type is computed, there need not exist a classification, but there could. If the discovery of computed types slows the operation down (especially with many possible types) it should be optional whether these are to be included. However, the supertypes of modeled types can be included at constant cost.

complete or incomplete Every type, regardless of whether it is computed or expressed is either complete or incomplete (with regard to an instance) and the engine can show this property for each of them as the check is not complex and redundant for computed types.

blueprint If blueprint information is present, it can easily be presented.

7.5.1.2 Feature Classification

Every feature of a clabject is either defined directly inside the clabject or is inherited from a supertype. In the former case it is called an eigenFeature. The combination is also possible because a feature can override another

one inherited from a supertype. The user may opt to explicitly render the inherited features and ask the question: "What effects does the deletion of a supertype (or a generalization) have on the features of the clabject?". Also, the features can be divided into those that are part of the type facade and those that are not (only features with a durability > 0 are part of the type facade).

7.5.1.3 Connection participation

Just like features, participation in a connection is inherited from supertypes. The reasoning engine can provide a list of pairs (`roleName`, `destination`) with the possible navigations for the source clabject. If desired, the pairs can be expanded to triples by including the realizing connection. Destination need not be a single clabject, but can be a set of clabjects as there may be multiple connections yielding the same roleName. Another customization is to filter only for navigable roles. Roles that are not navigable are not of interest to the clabject itself, but are part of its type facade so must not be neglected.

7.5.1.4 Supertypes

One operation is to compute the total set of supertypes. As with types, the engine can give information about each of them.

expressed or computed If the supertype is expressed, there is a generalization path between them. Unlike classification detection, generalization detection depends only on one model and is therefore faster to computate.

direct or indirect A supertype is direct if there exists a generalization connecting the subtype to the supertype. If the supertype is indirect, there is at least one intermediate supertype between them. Unlike complete and incomplete types, the direct and indirect supertypes

are logically equivalent to the clabject, so further properties are not defined.

7.5.1.5 Instance Facade

The instance facade captures the properties of a clabject. If a complete type was to be constructed from the clabject, its type facade would be the logical equivalent of the instance facade. The instance facade is the union of all features and roles, regardless of whether or not they are inherited.

7.5.1.6 Instances, Subtypes and Type Facade

Inverting the statements above, a clabject introspection service can also provide the following complementary information:

- All instances can be listed and classified according to whether they are computed or expressed and isonyms, hyponyms or offspring.
- All subtypes can be listed and classified according to whether they are computed or expresseds, direct or indirect.
- The type facade is not equivalent to the instance facade since for the type facade only those properties that would be passed upon instance creation are of relevance. Durability 0 features are not part of the type facade, neither are roles originating from potency 0 connections.

7.5.2 Correlation Introspection

An engine trying to discover new correlations should make use of the ones already present as much as possible. Let γ be the clabject in question and $|\Sigma_i.\mathsf{clabjects}|$ the number of clabjects on level i, with the same notation for the level's generalizations and classifications as well as the generalization's sub- and supertypes.

7.5.2.1 Generalization Checks

There are $(|\Sigma_{\gamma.\mathsf{level}}.\mathsf{clabjects}| - 1) * 2$ possible generalization relationships to γ. For each generalization ξ_i on level $\gamma.\mathsf{level}$, the number of possible generalization relationships for γ is lowered by either

- $(|\xi_i.\mathsf{supertype}| + |\xi_i.\mathsf{supertype}| - 1) * 2$ if γ is a part of ξ_i, or
- $|\xi_i.\mathsf{subtype}| + |\xi_i.\mathsf{supertype}|$ if γ is not a part of ξ_i.

If γ is a part of ξ_i, there is no need to investigate these pairs any further, but as γ can never be a sub- or supertype of itself, one of the participants of ξ_i (namely γ) has to be subtracted. If γ is not a part of ξ_i, all the subtypes will follow if γ is a supertype of one of the supertypes. Also, all the supertypes will follow if γ is a subtype of one of the subtypes. This is trivial for the case that γ is in ξ_i:

$$\gamma \in \xi_i.\mathsf{subtype} \implies \forall \gamma_s \in \xi_i.\mathsf{supertype} : \gamma.\mathtt{isSubtype}(\gamma_s)$$
$$\gamma \in \xi_i.\mathsf{supertype} \implies \forall \gamma_s \in \xi_i.\mathsf{subtype} : \gamma_s.\mathtt{isSubtype}(\gamma)$$

And if γ is not in ξ_i:

$$\exists \gamma_s \in \xi_i.\mathsf{subtype} : \gamma.\mathtt{isSubtype}(\gamma_s) \implies$$
$$\quad \forall \gamma_{s'} \in \xi_i.\mathsf{supertype} : \gamma.\mathtt{isSubtype}(\gamma_{s'})$$
$$\exists \gamma_s \in \xi_i.\mathsf{supertype} : \gamma_s.\mathtt{isSubtype}(\gamma) \implies$$
$$\quad \forall \gamma_{s'} \in \xi_i.\mathsf{subtype} : \gamma_{s'}.\mathtt{isSubtype}(\gamma_{s'})$$

So if γ is not a part of ξ_i it does not matter if γ is a subtype of one of the subtypes or a supertype of one of the supertypes. The point is that when searching for supertypes of γ it is sufficient to check the subtypes of ξ_i because if one of the subtypes of ξ_i is a supertype of γ, all the supertypes of the subtype in question will also be supertypes. The set of properties of a supertype is always a subset of the properties of the subtype. If the subtype's set of properties is sufficient to be a supertype for γ, a subset will also be sufficient and therefore the supertypes of ξ_i need not be checked.

For the search for subtypes of γ is analogous. If any supertype of ξ_i is a subtype of γ, its set of properties comprises the set of γ's properties. Any subtype will at least have the same set of properties, if not a larger one. The set in question will also have a super set of γ's properties and therefore be a subtype as well.

7.5.2.2 Classification Checks

There are $|\Sigma_{\gamma.\mathsf{level}+1}.\mathsf{clabjects}|$ possible instances of γ on the classified level. The classifications on $\Sigma_{\gamma.\mathsf{level}+1}$ contain information that the engine can process in order to have less classification pairs to check. The impact of one classification highly depends on the relations the type and instance are in:

- All the subtypes of the instance will be an instance of γ if the type is a subtype of γ.
- If the type is a subtype of γ, all the subtypes of the instance will be an instance of γ.

As the expressiveness of the classifications depends on the generalizations, it is advisable for the engine to check for inheritance relationships before going to the classifications of the next level. If the type is a subtype of γ, the type's properties are a super set of γ's properties. Through the existence of the classification (assuming it is valid) it is known that the instance satisfies the type properties of the type. As the instance satisfies the type properties of a super set of the properties of γ, the instance is also an instance of γ because γ's type properties are a subset of the type's properties. Any subtype of the instance has a super set of the properties of the instance's properties, so the subtype will also be an instance of both the type and γ.

$$\phi.\mathsf{type}.\mathtt{isSubtype}(\gamma) \implies$$
$$\phi.\mathsf{instance} \cup \{\gamma_i : \gamma_i.\mathtt{isSubtype}(\phi.\mathsf{instance})\} \subset \{\gamma_i : \gamma_i.\mathtt{isInstance}(\gamma)\}$$

If the type is a supertype of γ, there is a chance that the instance is an instance of γ as well, but no guarantee. The chance high since the set of type properties of γ is a super set of the type properties which the instance redefines. What is not guaranteed is that the instance redefines the delta of the type property sets of γ and the instance. Additional information can be drawn from the traits of the generalization. If it is complete, the instance must be an instance of either γ or one of the other subtypes of the type. If it is disjoint it can be an instance of at most one of the subtypes of the generalization.

Is there a way to eliminate an instance from the set of possible instances of γ? If γ and the types are not in an inheritance relationship (which the engine checks before starting to discover classifications) and the type is a complete type of the instance, then the instance is not an instance of γ. If the classification is not isonymic but hyponymic, the statement is not general because the instance may define all of γ's type properties in addition to those of the type. Also, if the type is a disjoint sibling of γ, the instance cannot be an instance of γ, but again the classification has to be isonymic.

It may seem like a lot of effort to subtract an instance from the

$$|\Sigma_{\gamma.\mathsf{level}+1}.\mathsf{clabjects}|$$

possible instances, but the checks presented are all independent of recursion and therefore much faster than the computation of a single classification.

7.5.2.3 potency computation

The potency of a clabject is input by the user but has a close relation to the artefacts in the ontology, since the potency equals the depth of the isonym tree. If the mode of modeling is switched from constructive to exploratory, the meaning of potency changes from the potential depth of the isonym tree to the actual depth. This is a point of failure as the actual depth may not always be correctly anticipated by the user or simply overlooked. The reasoning engine can compute the actual potency of the clabject and even

offer to correct it as an ontology cannot be complete with wrong potencies (see operation 13).

Operation 13 γ.`actualPotency`()

$classifiedModel \leftarrow \Sigma_{\gamma.\mathsf{level}+1}$
if $\neg classifiedModel$ **then**
  return 0
$isonyms \leftarrow \emptyset$
for $\gamma_i \in classifiedModel$.`clabject`() **do**
  if γ_i.`isIsonym`(γ) **then**
    $isonyms \leftarrow isonyms \cup \{\gamma_i\}$
if $isonyms = \emptyset$ **then**
  return 0
else
  return $1 + max(\gamma_i$.`actualPotency`()$\forall \gamma_i \in isonyms)$

7.5.3 Clabject Pair Introspection Service

While queries on singles clabjects gives information about all the relationships of a clabject, pair introspection just examines one of them.

7.5.3.1 Boolean Properties

The levels of conformance can be checked one by one. As each new level requires all of the earlier ones, an engine can break out of the computation once the answer is false and cache the positive results for later computations. The chain of checks is:

- Local Conformance
- Neighbourhood Conformance
- Multiplicity Conformance (in case of a pair of connections)
- Property Conformance

- No Additional Property Definition
- Potency Conformance
- Isonymic Classification
- Hyponymic Classification (here the chain of strict dependency ends. In fact, it is necessary and sufficient for hyponyms to property conform and define additional properties)
- Classification

The most interesting part of this chain of conformances is the reason why a certain part fails. The engine can then provide fixes to restore the property. These changes are not always possible to define. Table 7.1 gives an overview of the reasons for failures and possible fixes. Figure 7.4 shows the same information as a dependency graph.

7.5.4 Services about future states of an ontology

To answer questions about future states of the ontology, the reasoning engine requires as input:

1. The ontology in the original state,
2. The action to perform to produce the future state,
3. The property to check on the new state.

7.5.4.1 Adding a feature

The addition of a feature to γ will

- make γ a hyponym of all the types it was an isonym of,
- delete all isonymic classifications which γ is the type of, as the instances can no longer be isonyms,

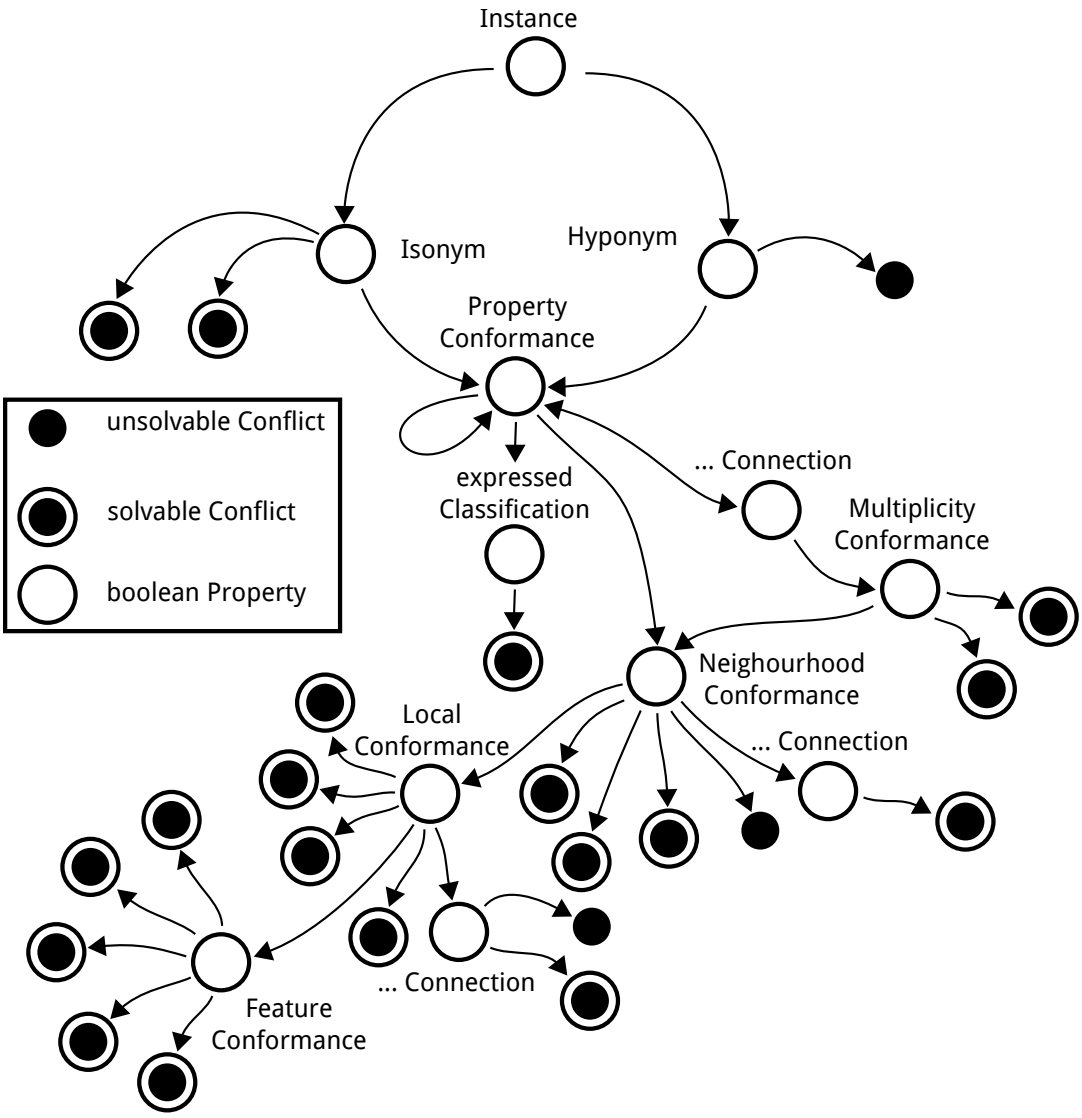


Figure 7.4: Dependencies between properties and ontological data including their resolvability

Table 7.1: Property dependencies, failure reasons and possible fixes

Property	Dependency	Failure Reason	Possible Fix
Feature Conformance		name	adjust
		datatype	adjust
		durability	adjust
		mutability	adjust
		value	adjust
Local Conformance	Feature Conf.	Feature Conformance	adjust
		missing feature	create
		level	transfer element
		wrong roleName	adjust
Connection		wrong navigability	adjust
		wrong order	
Neighbourhood Conformance	Local Conf.	missing roleName	
		roleName not local	adjust
		connection not local	adjust
		connection participant not local conforming	adjust
Connection		roleName not local	adjust
Multiplicity Conformance	Neighbour Conf.	too many	delete
		too few	raise upper
Expressed Classification		disjoint sibling	un-disjoint
Property Conformance	Neighbour Conf. Expr. Class. Property Conf.		
Connection	Multiplicity Conf. Property Conf.		
isIsonym	Property Conf.	potency mismatch	adjust
		too many features	delete
isHyponym	Property Conf.	too few features	
isInstance	isIsonym isHyponym		

- delete all classifications which a subtype of γ is the type of, as they too gain a new feature.

7.5.4.2 Deleting a connection

The deletion of a connection δ is followed by the deletion of all its roles as well. In addition, all connections that δ was a participant of are deleted. Correlations, which are now meaningless have to be deleted as well. This includes:

- any set relationships δ was a member of,
- any classification δ was a member of,
- all the generalizations which sub- or supertype set is empty without δ.

7.5.4.3 Marking a generalization disjoint/complete

The marking of a generalization makes a statement about the classified domain. So if a generalization ξ is now disjoint the following deadlock possibility arises: If there exists a clabject γ_i that holds classifications to $\xi.\mathsf{supertype}$ and $\gamma_s \in \xi.\mathsf{subtype}$ there may not exist another clabject $\gamma_{i'}$ holding classifications to $\xi.\mathsf{supertype}$ and another $\gamma_{s'} \in \xi.\mathsf{subtype}$. If such a situation is now present, the engine cannot solve it, but can only indicate that two pieces of information contradict each other and there is no judgement about which one is more important. Analogously, if ξ is marked as complete, any γ_i holding a classification to $\xi.\mathsf{supertype}$ may not hold another classification to $\gamma' \notin \xi.\mathsf{subtype}$.

After ξ is marked as disjoint or complete, the addition of a classification ϕ to a clabject γ_i already holding a classification to $\xi.\mathsf{supertype}$ may not be valid:

in the case of disjointness, if $\phi.\mathsf{type} \in \xi.\mathsf{subtype} \wedge (\exists \phi' : \phi'.\mathsf{instance} = \gamma_i \wedge \phi'.\mathsf{type} \in \xi.\mathsf{subtype} \wedge \phi'.\mathsf{type} \neq \phi.\mathsf{type})$.

in the case of completeness, if $\phi.\mathsf{type} \notin \xi.\mathsf{subtype}$.

Chapter 8

Ontology Evolution Services

Ontology evolution services provide mechanisms to automatically populate an ontology with new elements or to alter the state of the ontology based on user input. Ontology evolution services differ from the previous category of reasoning services e.g. by changing an ontologies state, adding new elements to the ontology or removing existing elements.

8.1 Subsumption

Subsumption means the detection of generalization relationships based on properties. With the `subsume` operation for clabjects the basic tool has already been established. Apart from creating the generalization elements, the real effort is to deduce the boolean attributes from the classified model.

Boolean generalization traits only make sense if the relation is not binary, so the subsumption algorithm tries to make as few generalizations connecting as many elements as possible. There may be cases where the desired behaviour is different, so the implementing tool may provide a preference to configure the creation of binary generalizations. The deduction of the boolean traits can be a very slow operation as a lot of classification checks are involved, so the tool may provide an option to configure that as well.

Operation 14 gives the formal definition of the subsumption algorithm. Every possible supertype is checked for all the subtypes that are possible. In that way, generalizations with one supertype and multiple subtypes will be created in favour of generalizations with multiple supertypes. The latter are used less frequently and have a weaker logical standing than the former.

Operation 14 Σ_i.subsumption()

for $\gamma_s \in \Sigma_i$.clabjects() **do**
  $subtypes \leftarrow \emptyset$
  for $\gamma \in \Sigma_i$.clabjects() $\setminus \{\gamma_s\}$ **do**
    // exclude γ_s or every clabject would become its own subtype
    if γ.subsume(γ_s) **then**
      $subtypes \leftarrow subtypes \cup \{\gamma\}$
  // the subtypes are discovered
  if $|subtypes| > 0$ **then**
    $\xi \leftarrow$ Generalization()
    ξ.supertype $\leftarrow \{\gamma_s\}$
    ξ.subtype $\leftarrow subtypes$
    $disjoint \leftarrow$ **true**
    $complete \leftarrow$ **true**
    for $\gamma_i \in \Sigma_{i+1}$.clabjects() : γ_i.isIsonym(γ_s) **do**
      if $\nexists \gamma_{s'} \in \xi$.subtype : γ_i.isIsonym($\gamma_{s'}$ **then**
        $complete \leftarrow$ **false**
      if $\exists \gamma_{s'}, \gamma_{s''} \subset \xi$.subtype : $\gamma_{s'} \neq \gamma_{s''} \wedge \gamma_i$.isIsonym($\gamma_{s'}$) $\wedge$ γ_i.isIsonym($\gamma_{s''}$) **then**
        $disjoint \leftarrow$ **false**
    ξ.disjoint $\leftarrow disjoint$
    ξ.complete $\leftarrow complete$

8.2 Refactoring

Refactoring generally means the tracing of a change to the affected artefacts without changing the semantics of the total concept. Models provide a suitable environment for general refactoring operations, as research on the topic shows (62). For potency-based multi-level modeling, the following concrete refactorings have been implemented:

Feature name If the name of a feature is changed in a type, every isonym of that type must change its corresponding feature. More specifically, it must change the feature with the old name to the new name as well. The isonym will always have a feature with the old name (requirement to be an instance) and will never have a feature with the new name (as it is an isonym it defines no more features than needed by the type and the type can only rename it to a name that is unique within its namespace).

Attribute durability/mutability The durability and mutability must be traced to the isonyms to maintain attribute conformance. If the durability and mutability is changed to * nothing needs to happen but the tool may be configured to change the isonyms attributes to * as well.

Clabject potency In the default use case, the potency of a clabject should only be lowered, as increasing the potency will most likely result in an incomplete ontology. The exception (also the default use case for manually increasing the potency) is that a new isonym has been created in the leaf model (the isonym tree is now deeper) and this change has been reflected in the artefacts. A tool can however provide an operation to update the potency to the actual depth of the isonym tree. Increasing a potency will be reflected in the isonyms in the same way as decreasing it will. In the default use case, the isonym tree will not have potency 0 clabjects when the potency is lowered. If it does,

those potencies cannot be decreased and will stay at 0. The classification between the already-zero clabject and the newly-zero clabject will be invalid and cannot revert to being a hyponym or an instance as it is not a hyponym (does not define additional properties) and is not an instance (neither hyponym nor isonym), so it has to be deleted. The tool should provide an undo operation in case such undesired side effects occur.

Role roleName/navigability As the role can be related to the (old) roleName in every isonym of the connection the change can be propagated.

In general, the goal is to restore any property of the ontology that might have been lost through the performed operation. In the type facade of a clabject, the changes need to be populated to the instances that would otherwise lose an isonymic relationship. In the instance facade the same principle would requires a change to a clabject's type. Although this may be desired, it is not the default use case. The default use case is to update the relationship to the type according to the new artefact. A tool may provide an option to define the desired behaviour. In the default use case, the recommended option to change the type facet as well would be to change the type directly, which would in turn trigger the change in the instance.

The direction for migrating refactoring changes is from types to isonyms. In a complete ontology it is to be expected that every instance has a complete type. More specifically, every clabject that is an instance has a complete type. So any hyponym is also an isonym of another type. If changes were also passed on to hyponyms as well, it is very likely that they would no longer be isonyms. Since isonymic classification is a stronger relationship than hyponymic classification the tool tries to preserve isonymic classification relationships. The problem only arises in the very rare case that the change comes from an incomplete type that does not affect the complete type.

If a type $\gamma_{complete}$ is a complete type of an instance $\gamma_{instance}$ and another type $\gamma_{incomplete}$ is an incomplete type, then $\gamma_{complete}$ is a subtype of $\gamma_{incomplete}$. As there exists an isonym of $\gamma_{complete}$ that is a hyponym of $\gamma_{incomplete}$ the type properties of $\gamma_{incomplete}$ must be a proper subset of the type properties of $\gamma_{complete}$. So the only case where a change coming from an incomplete type would not affect the complete type as well is when the complete type redefines or overrides the property, making it incompatible with the property after the refactoring change.

8.3 Model Instantiation

In constructive modeling mode, the *instantiate* operation is arguably the most important one. It is used to create new instances from the previously defined elements and recursively builds the whole model once the most abstract model is defined completely. If the top most model is sufficient, the instantiate operation is the only other operation used for element creation.

This constructional power has non-trivial formal requirements attached. As the name of the operation already implies, the resulting element has to be an instance of the type it has been constructed from. For traits and features, the definition is straightforward, but for roles it is not. Let γ' be the instance that resulted from the "instantiate" call to the type γ. For γ' to be an instance, γ' has to hold a role ψ' for every role ψ of γ. As roles cannot be defined directly but are derived from the existence of the defining connections, the "success" of the instantiation operation depends on the existence of an instance $\delta' : \delta'.isInstance(\delta)$ for all the δ in the roles ψ of γ.

Clearly the requirement cannot be ensured by an operation that only creates one new element. Of course, the operation could create an instance δ' for every δ, but then the operation would not only create an instance γ'

of γ but would instantiate possibly the whole model. While this might be desirable in some cases it can certainly not be the standard case.

8.3.1 Local Offspring

For this reason, the standard instantiate operation does not ensure that the check $\gamma'.instance(\gamma)$ evaluates to true after finishing the operation. Rather, it ensures that none of its actions negate a type/instance relationship. Formally, the instantiate operation can only ensure local conformance, as neighbourhood conformance already takes the connected clabjects into account and after the completion of the instantiate operation, γ' will not be connected to any other clabject. When instantiating a connection, the resulting connection will not even have participants. A tool could guess possible or even probable participants, but the automatic connection of elements unrelated to the executed operations would introduce new artefacts into the model. As the automatic creation of artefacts should in this case be avoided whenever possible, the instantiate operation will not connect any clabjects. Let γ, γ' be two clabjects

$$\gamma' := \gamma.instantiate(\dots) \implies \gamma'.\mathsf{potency} <_p \gamma.\mathsf{potency} \wedge$$
$$\gamma'.\mathsf{level} = \gamma.\mathsf{level} + 1 \wedge$$
$$\forall \eta \in \gamma.\texttt{features}() : \exists \eta' \in \gamma'.\texttt{features}() :$$
$$\eta'.\texttt{conforms}(\eta)$$

The definition does not differ between entities and connections, so there is no need to distinguish between the two. The above specification only shows the post conditions that the operation ensures in terms of describing the resulting instance. The definition of the instantiate algorithm is given below. A prerequisite for the instantiation of a clabject is the creation of conforming features. First, for attributes $\zeta := \zeta_t.\texttt{create}(\gamma)$:

Input: $\zeta_t.\mathsf{durability} > 0$

$\zeta \leftarrow$ `Attribute`

γ.children $\leftarrow \gamma$.children $\cup \{\zeta\}$
ζ.name $\leftarrow \zeta_t$.name
ζ.expressed $\leftarrow$ **true**// default explicit creation
ζ.datatype $\leftarrow \zeta_t$.datatype
ζ.value $\leftarrow \zeta_t$.value
if ζ_t.durability $= *$ **then**
  ζ.durability $\leftarrow \gamma$.potency
else
  ζ.durability $\leftarrow \zeta_t$.durability $- 1$
if ζ_t.mutability $= *$ **then**
  ζ.mutability $\leftarrow \zeta$.durability
else
  ζ.mutability $\leftarrow \zeta_t$.mutability $- 1$
return ζ

The equivalent for methods is alike $\pi := \pi_t$.`create`(γ):

Input: π_t.durability > 0
$\pi \leftarrow Method$
γ.children $\leftarrow \gamma$.children $\cup \{\pi\}$
π.name $\leftarrow \pi_t$.name
π.expressed $\leftarrow$ **true**// default explicit creation
π.body $\leftarrow \pi_t$.body
if π_t.durability $= *$ **then**
  π.durability $\leftarrow \gamma$.potency
else
  π.durability $\leftarrow \pi_t$.durability $- 1$
return π

Now with the *create* operations for the features defined, the instantiate operation for clabjects can be defined as well. As input parameters, the operation needs the model or clabject that will become the owner of the newly created one and the name to identify the instance. In case the instantiated clabject

is of potency * the tool can ask the user whether the instance shall be * as well or any given number. By default, a * potency will be passed on to the instance. $\gamma := \gamma_t.\texttt{instantiate}(\Sigma_i, name))$

Input: $\gamma_t.\mathsf{potency} > 0$

Input: $\gamma_t.\mathsf{level} = i - 1$

$\gamma \leftarrow Entity/Connection$

$\gamma.\mathsf{name} \leftarrow name$

$\Sigma_i.\mathsf{children} \leftarrow \Sigma_i.\mathsf{children} \cup \{\gamma\}$

$\gamma.\mathsf{expressed} \leftarrow \mathbf{true}$// default explicit creation

$\gamma.\mathsf{level} \leftarrow i$

for $\eta \in \gamma_t.\texttt{features}() : \eta.\mathsf{durability} > 0$ **do**

$\eta.\texttt{create}(\gamma))$

if $\gamma_t.\mathsf{potency} = *$ **then**

$\gamma.\mathsf{potency} \leftarrow *$

else

$\gamma.\mathsf{potency} \leftarrow \gamma_t.\mathsf{potency} - 1$

return γ

The algorithm also creates a classification between the two new elements, which may be configured by the user. There may be anonymous instances, so the name parameter is optional.

8.3.2 Participant Connection

After all the local offspring have been created, the roles can be created too. As a prerequisite for this operation, there has to exist a local offspring for every clabject model.

Let $\texttt{localOffspring}(\gamma)$ be the operation to retrieve the local offspring created from γ and Σ_i be the model subject to the operation. The algorithm to create the roles is as follows:

for $\delta \in \Sigma_i.\texttt{connections}()$ **do**

for $\psi \in \delta$.`roles`() **do**
  ψ' ← `Role`()
  ψ'.roleName ← ψ.roleName
  ψ'.navigable ← ψ.navigable
  if `localOffspring`(δ).potency > 0 **then**
    ψ'.lower ← ψ.lower
    ψ'.upper ← ψ.upper
  ψ'.destination ← `localOffspring`(ψ.destination)

8.3.3 Populating the new Model

Now that the algorithms to create the artefacts and connect them are available, a valid classified model can be created.

`localOffspring` ← $\emptyset$ // the data structure to store the new elements
for $\gamma \in \Sigma_i$ **do**
  if $\gamma \notin$ `localOffspring` **then**
    // γ does not yet have a local offspring
    `localOffspring`.`add`($\gamma \rightarrow \gamma$.`instantiate`($\Sigma_{i+1}$)) // whether or not the whole set or just γ is created doesn't matter as long as `localOffspring` is administered
`connectParticipants` // as described in 8.3.2

After the connection of the participants, the artefacts are complete. The creation of correlations is therefore optional.

8.3.4 Multiplicity Satisfaction

After the participants have been connected the classified model is complete except for multiplicity conformance. A complete model would required that the instances pass the test they have been designed to pass, namely to be offspring of the types they were created from.

After each type has been instantiated, there are exactly as many clabjects in the classified model as in the classifying model. Depending on the multiplicities of the present connections, the created domain might not be able to satisfy these constraints. If the constraints are not met, additional clabjects are required to define and participate in the connections. Existing clabjects can participate in connections as well. The tool cannot automatically decide how many clabjects to create and which ones to connect. The only operation which is possible and may be configured is to create the local offspring of the connections, so that the user knows how many connections are needed to fulfil the constraints.

8.3.5 Correlation Creation

By definition, the local offspring operation creates a classification between each created clabjects and its blueprint. If the clabjects in the classified model have potency zero, generalizations cannot exist in the model. Even if the potency is greater than zero, generalizations are not necessary as the local offspring creation creates all the necessary properties within the clabjects that require them. This does not mean generalizations are meaningless. The operation to detect generalization and delete redundant properties can still be executed. The set relationships can simply be copied from the classifying model to the classified model.

8.3.6 Classifying Model Creation

Once the instantiation of types has been addressed, the complementary operation to construct a type from a given instance needs to be addressed as well. The process is essentially the same, just the direction is inverted: from the instance facade of a clabject a locally complete type is created. The big difference is that the number of types is usually smaller than the number of instances. So, creating a type for every instance clabject will probably lead

to some redundant types. Formally, a type is redundant if there is another equal one. Multiple equalities can be avoided in one of two ways:

1) prior to creating a new type, the tool checks if the type has already been created from a clabject with the same instance facet,

2) after all the types have been created, redundant ones are deleted.

Each of the two approaches has its pros and cons. With 1) the overall computational complexity is lower as searching for equal clabjects is expensive. The downside is that the checking for an equal instance facade can be just as expensive and the instance domain is usually larger than the type domain. 2) has the benefit of a straightforward implementation as the notion of clabject equality is established. However, it has to do some extra work to receive the correct multiplicity values from the connections which can simply be counted in the other scenario.

8.3.6.1 Local Complete Type

Input: γ

$\gamma_t \leftarrow Entity/Connection$
$\gamma_t.\mathsf{potency} \leftarrow \gamma.\mathsf{potency} + 1$
$\gamma_t.\mathsf{level} \leftarrow \gamma.\mathsf{level} - 1$
$\gamma_t.\mathsf{expressed} \leftarrow \mathbf{true}$
if $\gamma \in \mathsf{Connection}$ **then**
  $\gamma_t.\mathsf{transitive} \leftarrow \mathbf{false}$
for $\eta \in \gamma.\texttt{features}()$ **do**
  $\eta_t \leftarrow Attribute/Connection$
  $\eta_t.\mathsf{name} \leftarrow \eta.\mathsf{name}$
  $\eta_t.\mathsf{durability} \leftarrow \eta.\mathsf{durability} + 1$
  $\gamma_t.\mathsf{children} \leftarrow \gamma_t.\mathsf{children} \cup \{\eta_t\}$
  if $\eta \in \mathsf{Method}$ **then**
    $\eta_t.\mathsf{body} \leftarrow \eta.\mathsf{body}$

η_t.input $\leftarrow \eta$.input

η_t.output $\leftarrow \eta$.output

else

η_t.datatype $\leftarrow \eta$.datatype

η_t.value $\leftarrow \eta$.value

η_t.mutability $\leftarrow \eta$.mutability $+ 1$

For potency and mutability values, the tool can only handle the standard case. If the mutability of a type attribute needs to be zero, the user has to manually set this feature. The same goes for star potencies.

8.3.7 Connection participation and multiplicities

Connecting participants is handled in the same way as with the instantiation of types. From a map of the newly created elements the destinations are registered with their roles. Navigability is copied accordingly (as well as the roleName). The interesting part is the multiplicities of the roles. After all the connections have been processed the multiplicities of the roles can be determined based on various criteria. These include but are not limited to:

Always [0..*] The loosest possible multiplicity. In fact, in this case there does not need to be any adjustment from the default. The only benefit of the operation is to remove redundant connections. Here it is possible to add/remove as many connection instances as desired.

[**0..actual**] There can be no connections added but removal is possible.

[**actual..actual**] No connection instance can be added or removed.

The following algorithm checks for redundant connections, removes them and administers the map of instances to types accordingly. The destination of the roles have already been linked to the types.

Input: `localOffspring` // map of a type to isonyms

Input: Σ_i // the model the types reside on

```
for γ ∈ Σ_i.clabjects() do
  for γ' ∈ Σ_i.clabjects() do
    if γ.equals(γ') then
      localOffspring[γ] ← localOffspring[γ] ∪ localOffspring[γ']

// now the redundant connections are removed
for δ ∈ Σ_i.connections() do
  for ψ ∈ δ.roles() do
    min ← ∞
    max ← 0
    for γ_i ∈ localOffspring[ψ.destination] do
      actual ← |{ψ' : ψ'.connection ∈ localOffspring[δ]
          ∧ψ'.roleName = ψ.roleName
          ∧ψ'.destination = γ_i}|
      if actual < min then
        min = actual
      if actual > max then
        max = actual
    // assuming the strictest multiplicity possible
    ψ.lower ← min
    ψ.upper ← max
```

8.4 Establishing a property

As table 7.1 shows, there can many reasons why a particular property might not apply to a clabject and various strategies exist for making the property hold. If a certain property does not hold, a tool can try to alter the clabject to satisfy that property. The tool can neither invent artefacts nor guarantee the property, but it usually can indicate where the problem lies and identify what needs to be changed. Each algorithm follows the same schema: It

looks for possible reasons preventing the property from holding if it finds any [1] it applies the corresponding changes or identifies the reason for the problem.

The side effects of the performed changes are neither monitored nor considered. The operation cannot ensure that changes to one feature does not cause other relationships to become invalid.

8.4.1 Feature Conformance

Normally feature conformance is defined between two features, but in terms of boolean properties feature conformance means that an instance does not redefine a feature required by the type.

Input: η_t, γ_i // the type feature that is not found at γ_i
  for $\eta \in \gamma_i$.`features`() **do**
    if η.name $= \eta_t$.name **then**
      // the feature is present, check why it does not conform
      if η.datatype $\neq \eta_t$.datatype **then**
        η.datatype $\leftarrow \eta_t$.datatype
      if η_t.durability $\neq * \wedge \eta_t$.durability $- 1 \neq \eta$.durability **then**
        η.durability $\leftarrow \eta_t$.durability $- 1$
      if η_t.mutability $\neq * \wedge \eta_t$.mutability $- 1 \neq \eta$.mutability **then**
        η.mutability $\leftarrow max(0, \eta_t$.mutability $- 1)$
      if η_t.mutability $= 0 \wedge \eta_t$.value $\neq \eta$.value **then**
        η.value $\leftarrow \eta_t$.value
      return
  $\eta_i \leftarrow newFeature$ // the feature is not present, create it
  eta_i.name $\leftarrow \eta_t$.name
  η_i.datatype $\leftarrow \eta_t$.datatype
  η_i.durability $\leftarrow \eta_t$.durability $- 1$
  η_i.mutability $\leftarrow max(0, \eta_t$.mutability $- 1)$

[1] which may not always be possible

η_i.value $\leftarrow \eta_t$.value
return

This algorithm is for attributes, but the name and durability of methods can be handles in the same way.

8.4.2 Local Conformance

Missing feature conformance is already covered in the previous paragraph. For entities the only thing left is level mismatches. Connections can correct false role information at one end. If more than one role has misleading information, the tool will work out which one to adjust using which pattern.

Input: γ_t, γ_i // γ_i shall local conform to γ_t
if γ_t.level $+ 1 \neq \gamma_i$.level **then**
return Elements need to be moved to different levels // no element moves across levels
// The following only applies to connections
if γ_i.`order`() $\neq \gamma_t$.`order`() **then**
return connections are not of same order // no actions on different orders
for $\psi_t \in \gamma_t$.`roles`() **do**
if $\nexists \psi_i \in \gamma_i$.`roles`() : ψ_t.roleName $= \psi_i$.roleName **then**
if $\neg wrongRole$ **then**
$wrongRole \leftarrow (\psi_t, \psi_i)$
else
return more than one roleName mismatch // too little common information
else
if ψ_t.navigable $\neq \psi_i$.navigable **then**
ψ_i.navigable $\leftarrow \psi_t$.navigable // navigability adjusted
if $wrongRole$ **then**

$wrongRole[1]$.roleName $\leftarrow$ $wrongRole[0]$.roleName

$wrongRole[1]$.navigable $\leftarrow$ $wrongRole[0]$.navigable // wrong role updated

8.4.3 Neighbourhood Conformance

Neighbourhood conformance depends on local conformance and introduces few additional requirements other than requiring local conformances. So the possible strategies for ensuring neighbourhood conformance are based on ensuring local conformance.

Input: γ_t, γ_i // γ_i shall neighbourhood conform γ_t

if $\neg\gamma_i$.`localConforms`(γ_t) **then**

try to ensure γ_i.`localConforms`(γ_t)

for $\psi_t \in \gamma_t$.`navigations`() **do**

if $\nexists\psi_i \in \gamma_i$.`navigations`() : ψ_i.roleName = ψ_t.roleName **then**

missing roleName // cannot be fixed automatically

else

$\psi_i \leftarrow \gamma_i$.`navigations`() : ψ_i.roleName = ψ_t.roleName

if $\neg\psi_i$.destination.`localConforms`(ψ_t.destination) **then**

try to ensure ψ_i.destination.`localConforms`(ψ_t.destination) // navigation not local

if $\neg\psi_i$.connection.`localConforms`(ψ_t.connection) **then**

try to ensure ψ_i.connection.`localConforms`(ψ_t.connection) // connection not local

// the rest applies only to connections

for $\psi_t \in \gamma_t$.`roles`() **do**

$\psi_i \leftarrow \gamma_i$.`roles`() : ψ_i.roleName = ψ_t.roleName

if $\neg\psi_i$.destination.`localConforms`(ψ_t.destination) **then**

try to ensure ψ_i.destination.`localConforms`(ψ_t.destination) // participant not local

8.4.4 Multiplicity Conformance

Multiplicity conformance only applies to connections. It depends on neighbourhood conformance, but cannot fix it automatically. The set of connections which validate the multiplicity is determined through neighbourhood conformance, but if the set does not satisfy the constraint, it is up to the user to decide whether the model or the constraint needs to be adjusted.

Input: δ_t // the connection to process the multiplicity constraint for

$model \leftarrow \Sigma_{\delta_t.\mathsf{level}+1}$

$domain \leftarrow model.\texttt{clabjects}()$

$actualLower \leftarrow \infty$

$actualUpper \leftarrow 0$

// construct the actual multiplicity, see 6.4

for $\psi \in \delta.\texttt{roles}()$ **do**

  for $\psi' \in \delta.\texttt{roles}() : \psi \neq \psi'$ **do**

    for $\gamma \in domain : \gamma.\texttt{neighbourhoodConforms}(\psi'.\mathsf{destination})$ **do**

      $actuals \leftarrow \psi'' \in model.\texttt{roles}() :$

        $(\psi''.\mathsf{destination} \neq \gamma) \wedge$

        $(\gamma \in \psi''.\mathsf{connection}.\texttt{participants}()) \wedge$

        $(\psi''.\mathsf{roleName} = \psi.\mathsf{roleName}) \wedge$

        $(\psi''.\mathsf{connection}.\texttt{neighbourhoodConforms}(\delta))$

      $actualLower \leftarrow min(actualLower, |actuals|)$

      $actualUpper \leftarrow max(actualUpper, |actuals|)$

  // adjust ψ's bounds if necessary

  $\psi.\mathsf{lower} \leftarrow min(actualLower, \psi.\mathsf{lower})$

  $\psi.\mathsf{upper} \leftarrow min(actualUpper, \psi.\mathsf{upper})$

8.4.5 Expressed Classification

Expressed classification does not depend on any of the previous properties. Expressed classification is easy to achieve since the only statement that needs to be changed is the boolean trait of the generalization. If this boolean trait is the only thing preventing property conformance, it probably is the right choice to change the generalization (as there are artefacts indicating it is wrong). Generally, only the user can decide which piece of information is more important: the trait of the generalization or the artefacts in the classified domain. It is quite possible that the trait is there to prevent the property conformance the tool is trying to enable by changing the generalization.

Input: γ_i, γ_t // After the operation, γ_i shall not be an expressed instance of a disjoint sibling of γ_t

$classGener \leftarrow \{\xi \in \Sigma_{\gamma_t.\mathsf{level}} : \xi.\mathsf{disjoint}\}$ // all disjoint generalizations on the type level

$possibles \leftarrow \{\gamma_t\} \cup \gamma_t.\texttt{modelSupertypes}()$ // the types that any instance of γ_t is an instance of

$excluded \leftarrow \emptyset$

$insts \leftarrow \{\phi \in \Sigma_{\gamma_i.\mathsf{level}} : \phi.\mathsf{expressed} \wedge \phi.\mathsf{instance} = \gamma_i\}$ // all the expressed classifications γ_i is the instance of

for $\xi \in classGener$ **do**

  if $possibles \cap \xi.\mathsf{subtype} \neq \emptyset$ **then**

    // if now any of the classifications has one of the subtypes (not one of possibles) as type, ξ ensures expressed classification

    for $\phi \in insts$ **do**

      if $(\phi.\mathsf{type} \notin possibles) \wedge$

      $((\phi.\mathsf{type} \in \xi.\mathsf{subtype})$

      $\vee(\phi.\mathsf{type}.\texttt{getModelSupertypes}() \cap \xi.\mathsf{subtype} \neq \emptyset))$ **then**

        $\xi.\mathsf{disjoint} \leftarrow$ **false**// ξ has to be switched

Formally, the algorithm is very similar to the one for detecting expressed classification (see operation 12), and if it is detected, the reason for it is negated. So this algorithm does neither check that afterwards γ_i property conformsto γ_t nor does it weigh the generalization trait against the artefacts. The only thing it does is ensure that afterwards the check for expressed classification will return false. All other decisions have to be taken before running the operation.

8.4.6 Property Conformance

Property conformance not only depends on the two subject clabjects, but all connected clabjects as well. An automated attempt to ensure property conformance does not process all the connected clabjects, but only the two subject ones. The reason is that the operation should not have any side effects on portions of the model other than what the user selected.

As a consequence, the operation cannot ensure property conformance, because only a portion of the defining clabjects can be altered. What the operation can ensure is that these two subject clabjects do not prevent property conformance.

Input: γ_i, γ_t // the (possible) instance-type pair to check
  if $\neg\gamma_i$.`neighbourhoodConforms`(γ_t) **then**
    fix neighbourhood conformance
  if γ_i.`isInstanceOfExcluded`(γ_t) **then**
    fix expressed classification
  // the following applies only to connections
  if $\neg\gamma_t$.`multiplicityConformance`() **then**
    fix multiplicity conformance

8.4.7 Isonymic Classification

Ensuring that a clabject is an isonymic instance of a type is relatively straightforward once property conformance has been attained for each of

the types attributes. The only two things that can then stand in the way of isonymic classification is that there is a potency mismatch or the clabject has more attributes than are required by the type. Again, if there is such a mismatch it is up to the user to decide which attributes to delete or which potency to change.

Input: γ_i, γ_t // γ_i shall be the isonym of γ_t
  if $\neg\gamma_i.\texttt{propertyConforms}(\gamma_t)$ **then**
    fix property conformance
  if $(\gamma_t.\mathsf{potency} \neq *) \wedge (\gamma_i.\mathsf{potency} + 1 \neq \gamma_t.\mathsf{potency})$ **then**
    if $\gamma_t.\mathsf{potency} = 0$ **then**
      // (assumed) wrong type potency
      $\gamma_t.\mathsf{potency} \leftarrow 1$
      $\gamma_i.\mathsf{potency} \leftarrow 0$
    else
      $\gamma_i.\mathsf{potency} \leftarrow \gamma_t.\mathsf{potency} - 1$ // (assumed) wrong instance potency
  // from now on property conformance is assumed
  for $\eta_i \in \gamma_i.\texttt{features}()$ **do**
    if $\nexists\eta_t \in \gamma_t.\texttt{features}() : \eta_i.\mathsf{name} = \eta_t.\mathsf{name}$ **then**
      // if γ_t has a feature of the same name, they conform
      delete η_i
  for $\psi_i \in \gamma_i.\texttt{navigations}()$ **do**
    if $\nexists\psi_t \in \gamma_t.\texttt{navigations}() : \psi_t.\mathsf{roleName} = \psi_i.\mathsf{roleName}$ **then**
      // a single role cannot be deleted, the connection has to be deleted
      delete $\psi_i.\mathsf{connection}$

8.4.8 Hyponymic Classification

As with isonymic classification, ensuring that a clabject is an isonymic instance of a type is relatively straightforward once property conformance has been attained for each of the type's properties. In fact, since potency conformance is not required for hyponymic classification, if prop-

erty conformance hold for each of the properties of the type, the only thing that can stop a clabject from being a hyponym of it is that it is an isonym. The change needed to make it a hyponym is therefore simply to add another property. In general it does not make sense for the tool to do this, but for the user. The tool can however indicate what change is needed.

Input: γ_i, γ_t // γ_i shall become a hyponym of γ_t
  if $\neg\gamma_i$.`propertyConforms`(γ_t) **then**
    try to fix property conformance
  // property conformance is assumed
  if $\neg\gamma_i$.`hasAdditionalProperties`(γ_t) **then**
    define additional property // isonym?

8.4.9 Instance Relationship

Instance is an umbrella concept for isonymic and hyponymic classification. As a consequence, the things that can stop a property conforming clabject from being an instance of a type is that it is an "isonym with the wrong potency". A tool cannot decide whether the most appropriate solution is to add an additional property (to make it a hyponym) or to correct the potency (to make is a proper isonym).

Input: γ_i, γ_t // γ_i shall become an instance of γ_t
  if $\neg\gamma_i$.`propertyConforms`(γ_t) **then**
    try to fix property conformance
  // property conformance is assumed
  if $\neg\gamma_i$.`hasAdditionalProperties`(γ_t) **then**
    return fix potency by trying to fix isonym
  else
    return **true**// property conformance was the problem. γ_i is now a hyponym

8.4.10 Remove redundant Generalizations

A redundant generalization is a generalization that does not introduce any new information except for its boolean traits. All properties of the supertype(s) are inherited by the subtype(s). If the generalization does not define any new properties for the subtypes, it is redundant because all the information it brings to the subtypes is already there. If the generalization does not hold any boolean traits under such circumstances it is completely redundant and can be deleted. If it holds boolean traits it does add some information to the ontology. A tool may provide an option to delete all redundant generalizations or delete them on a case-by-case basis according to the wishes of the user. A generalization can also be redundant the inheritance it defines is already defined by (one or more) other generalizations — that is, if all the subtypes can reach all the supertypes via other generalizations. The following algorithm detects this second kind of redundancy.

Input: ξ // the generalization to be checked for redundancy
  $toFind \leftarrow \xi$.supertype
  $source \leftarrow \xi$.subtype
  $edges \leftarrow \Sigma_{\xi.\texttt{level()}}.\texttt{generalizations}() \setminus \{\xi\}$
  for $\gamma_s \in source$ **do**
    $found \leftarrow \emptyset$
    $queue \leftarrow \{\gamma_s\}$
    while $queue \neq \emptyset$ **do**
      $current \leftarrow queue$.pop()
      for $edge \in edges : current \in edge$.subtype **do**
        $queue \leftarrow queue + edge$.supertype
      $found \leftarrow found \cup \{current\}$
    if $toFind \not\subset found$ **then**
      return false// ξ contains unique information

return true// every inheritance is also found through other generalizations

The algorithm does not check for inherited properties, just inheritance in general.

8.4.11 Remove redundant Features

Features that are inherited from supertypes can also be directly modeled at the levelof the subtype. The complementary operation is to remove directly defined features that are not needed as they are already implied through inheritance. While performing this check it is important not to delete features that override the inherited ones.

Input: γ // the clabject to remove the redundant features from

$donors \leftarrow \gamma$.`getModelSupertypes`()

for $\eta_i \in \gamma$.`eigenFeatures`() **do**

for $donor \in donors$ **do**

if $\exists \eta_t \in donor$.`eigenFeatures`() : η_i.`equals`(η_t) **then**

delete η_i

Chapter 9

Case Studies

The case studies presented in this chapter serve two purposes. First, they illustrate the concepts presented in the earlier chapters with well known examples from the realms of software engineering and knowledge engineering. Secondly, they demonstrate that the introduced formalisms are able to support the same operations that these examples were designed to illustrate. The case studies have been carefully chosen to include well known examples from the software engineering community and the knowledge engineering community.

9.1 The Pizza Ontology

The Pizza ontology is a very famous example taken from the tutorial(40) of the OWL flagship tool Protégé(43). It features different ingredients of the famous dish and explains the capabilities of the tool through various examples. Part of the example's success is its relation to an everyday domain that everybody immediately can relate with.

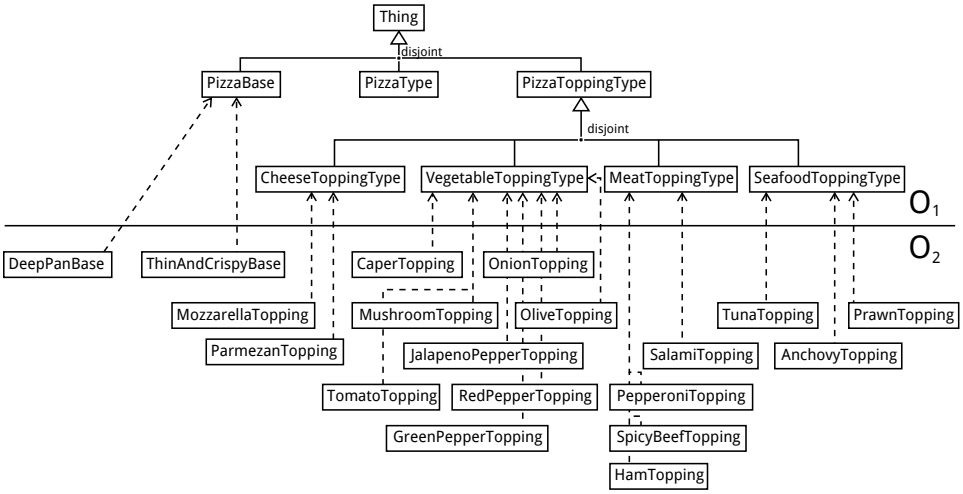


Figure 9.1: The asserted class hierarchy of the pizza ontology

9.1.1 Defining the Toplevel Classes

Initially the top level classes are defined: Pizza, PizzaBase and PizzaTopping. PizzaBase and Topping are superclasses for more special types, like CheeseTopping, TunaTopping or DeepPanBase. In multi-level modeling we can distribute those concepts over two ontological levels. CheeseTopping and MeatTopping are different kinds of toppings, MozarellaTopping is one instance of the topping kind (or type) CheeseTopping. PizzaToppingType is the common supertype for the different topping kinds. The same holds for PizzaBase, except there is no further distinction of the base kinds and the direct subclasses (here in multi-level modeling instances) are the concrete kinds DeepPanBase and ThinAndCrispyBase. The basic ontology is shown in figure 9.1.(13)

The naming of the clabjects in O1 differs from the tutorial in that the word *type* is appended to some of the concepts. This has been done to better reflect the correct level of ontological ordering between the concepts. If a new clabject in O2, MargharitaPizza is introduced the natural language construct “Margharita is a PizzaType” works well. If now on O3 a clabject

called ExampleMargharita is created, the sentence "ExampleMargharita is a Margharita" still works well while the sentence "ExampleMargharita is a PizzaType" does not work.

As the concept of generalization is not present and the inheritance is woven into the definition of classes in a subclass-tree, there are no generalization elements to carry disjointness information. This information has to be entered in the window of the class. In PMLM the subtypes of toppings are made disjoint by setting the generalization to disjoint.

9.1.2 OWL Object Properties

Object properties are relationships between objects. Protégé defines these properties independently from the objects they connect. This is very much the same behavior as with connections. They can be created independently from their participating clabjects. The ends of the properties are called domain and range in OWL. These directly map to the participating clabjects in PMLM. In OWL, every object property is binary and the direction goes always from the domain to the range. Therefore, the name of the property is rather the roleName identifying the destination for the source. So an equivalent PMLM connection would be anonymous with the property name as the navigable destination roleName. OWL Properties can have certain specific properties:

Inverse Rather than a special label to one property, a property can be named to be the inverse property of another. The PMLM equivalent is to create an inversion between the two inverse clabjects.

Functional A functional property can connect an individual only to one other individual. The equivalent in PMLM is to have a multiplicity of 1 at the destination role.

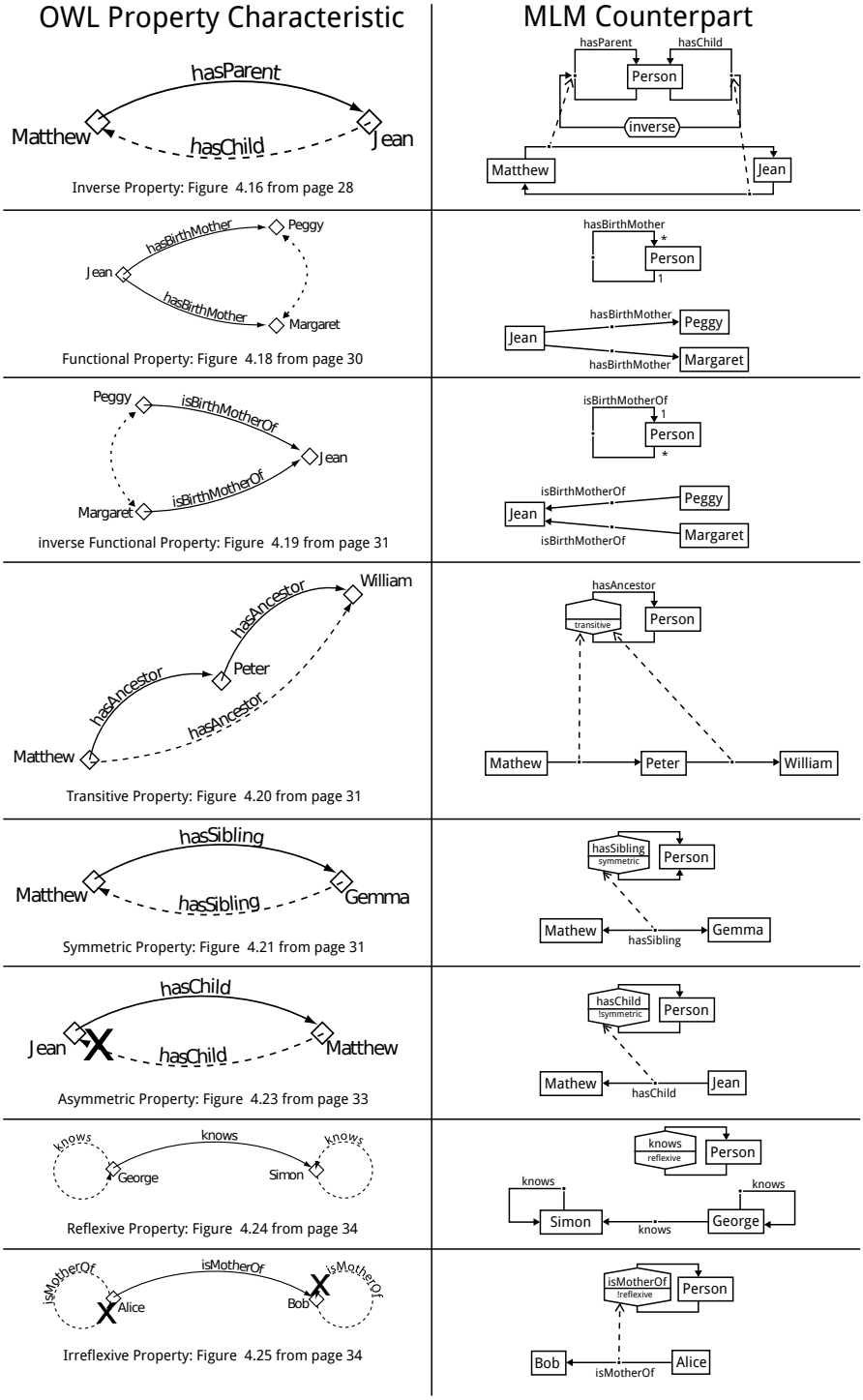


Figure 9.2: OWL property characteristics and their counterpart representation in PMLM. The pictures on the left are screenshots taken from (40).

Transitive A transitive property can infer the existence of other instances from a certain pattern of already present instances. The PMLM supports *transitive* connections as well with the same semantics.

Symmetry A symmetric property is a property that is implicitly always a pair of directed properties connecting the source to the target and vice versa. The PMLM equivalent is a connection with two navigable roles. As the roleNames have to be unique in PMLM the mapping is not exact. It is however possible to also define two directed connections with the same navigable roleName. If the symmetric trait of the connection is set, the classified domain has to ensure symmetry, whether it be by multiple connections or a naming convention for the roleName.

Asymmetry An asymmetric property states that if $A.B$ is true then $B.A$ cannot be true. The statement is quite strict as it states what is not possible. In a closed world assumption, everything that is not asserted does not exist. But even for a closed world ontology the negation of symmetry holds information, namely what is not possible to define. As the PLM supports three values for the symmetry trait (true, false and unset), the negation of symmetry states that for every navigation the counter navigation from the target back to the source cannot exist.

Reflexion A reflexive property always connects to itself. So every individual in the domain is also in the range of the property. If a connection is reflexive, the classified domain is only consistently classified if it contains all the required connections.

Irreflexive An irreflexive property states that no individual can be connected to itself via the property. In PMLM it is the same as with symmetry. If reflexive is set to false, the classified domain must not contain any reflexivity.

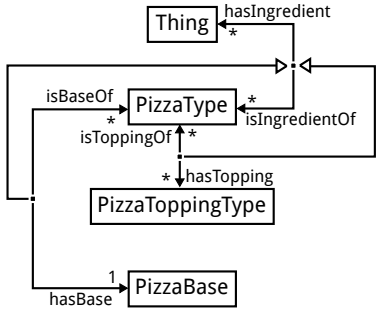


Figure 9.3: The object properties of the pizza ontology in the multi-level modeling representation.

Figure 9.2 shows the different characteristics and their respective representation in PMLM.

The pizza ontology defines three object properties: hasIngredient, hasTopping and hasBase each of which has inverse properties isIngredientOf, isBaseOf and isToppingOf. The hasIngredient and hasTopping property are many-to-many relations, the hasBase is many-to-one. The hasTopping and hasBase properties are subtypes of the hasIngredient property. As subtypes can only get stricter in their multiplicities, the hasIngredient property has to have many to many multiplicity. In PMLM, the domain and range of the properties have to be specified. The domain for all of the properties is PizzaType. The range of hasBase is PizzaBase, for hasTopping it is PizzaToppingType. So the range for the common supertype hasIngredient has to be a supertype of PizzaBase and PizzaToppingType. As there is no such class as PizzaIngredient, the only valid supertype is Thing. The inverse properties can be achieved by making the connections navigable in both directions and reflecting the property names with the roleNames of the participants. Figure 9.3 shows the connections presented in PMLM that show the facts expressed in the tutorial.

9.1.3 Defining classes through property restrictions

A class can be described through property and/or existential restrictions. These classes can either be anonymous or then given a name based on their definition. The characteristic feature of these classes is that the set of its instances depends on the formal description and will change according to it.

While the current prototype tool does not support the definition of clabjects by entering restrictive expressions, the evaluation of expressions is supported. What is missing therefore to support this feature is to allow the definition of a type by evaluating the set of its instances based on a selection expression. Going further, this type can then be described in terms of its domain properties. What is already supported, however, is modeling new types. So the restrictions are not input via an expression, but are expressed through artefacts.

9.1.4 Named pizzas

MargharitaPizza is modelled as a Pizza with two connections: One hasTopping to TomatoTopping and one hasTopping to MozzarellaTopping. The inference engine will then infer that MargharitaPizza is an instance of PizzaType as the connections are instances of the hasTopping connection. By the same principle, AmericanPizza, HotAmericanPizza and SohoPizza can be modelled. Figure 9.4 shows the model of the named pizzas.

9.1.5 Special types of pizza.

After the named pizzas there are some more specializations of the types a pizza can be an instance of:

VegetarianPizzaType is a pizza with only Cheese and Vegetable topping. Vegetarian pizza cannot be a subtype of PizzaType, because the hasTopping connection from PizzaType to PizzaToppingType allows MeatToppingTypes. As any subtype would inherit that connection,

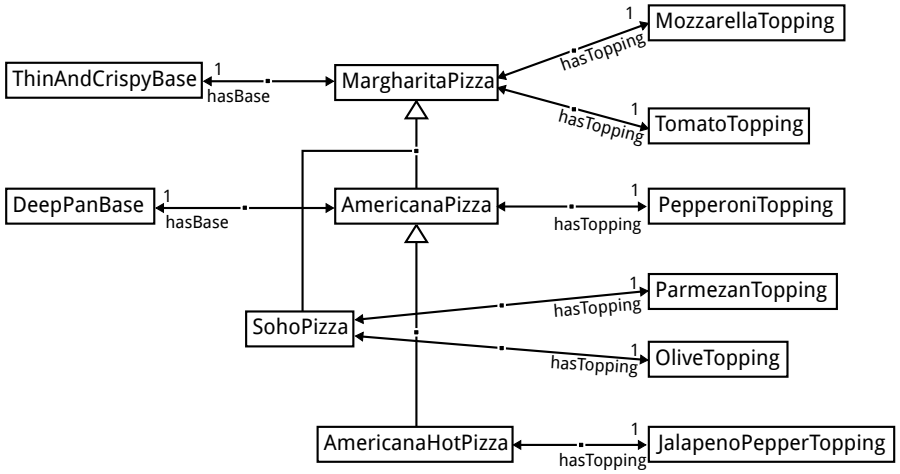


Figure 9.4: The named pizzas from the pizza ontology

vegetarian pizza would also inherit that possibility. Of course, the connection could also be constrained at the subtype.

InterestingPizzaType is a pizza with at least three toppings. This new type has to override the connection and be more strict on the multiplicities.

CheesyPizzaType is any pizza with a cheese topping.

High/Low Calorie Pizza is a pizza that does not exceed low or high calorie thresholds. The modeling is achieved through the mutability feature. The types set the value of the attribute to an expression that evaluates to true or false. Its mutability is then set to zero, meaning instances cannot redefine that feature. This definition is extended so that the instances may not redefine the value to a value that evaluated the expression to false. The calorie attribute is introduced at level 2 and not 1. On level 1, the corresponding type would by PizzaType. But a PizzaType does not have a discriminable calorie value. Neither do pizza types on level 2, but they can constrain the value of their

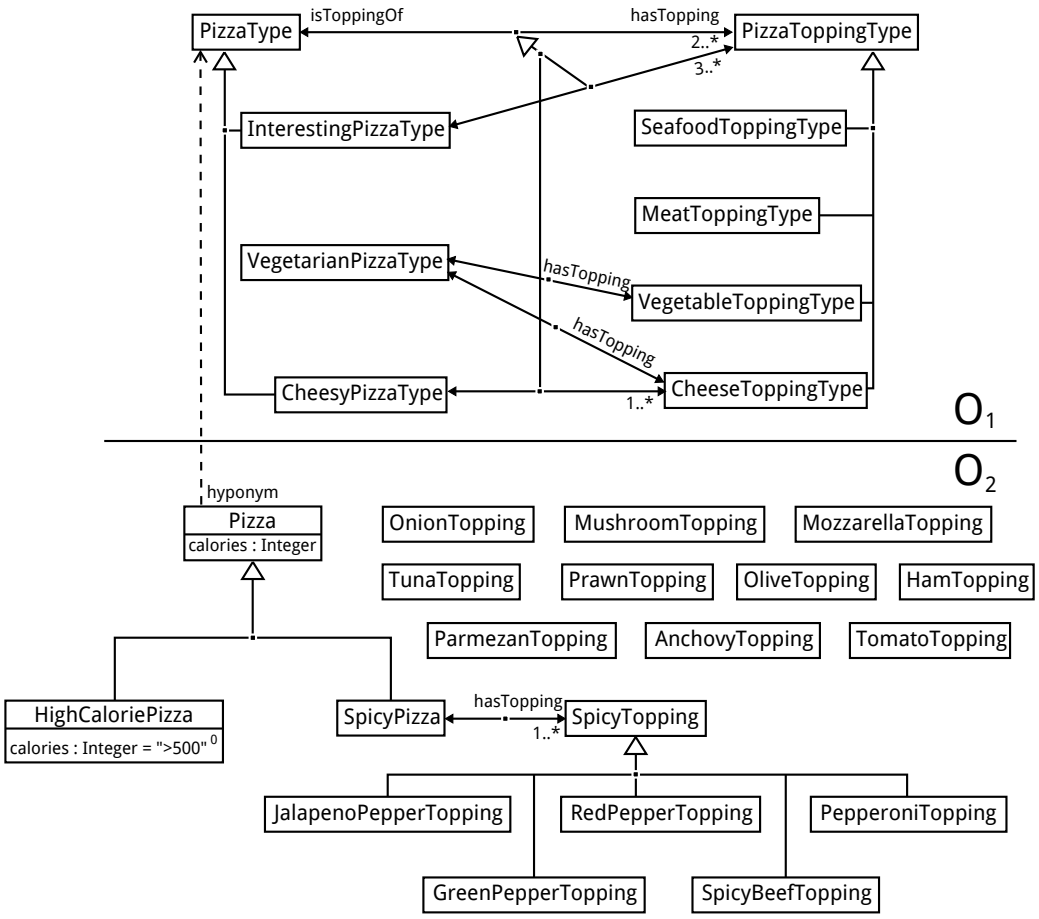


Figure 9.5: The special pizza types in the pizza ontology

instances (like low/high calorie pizza) or just state that it exists at level 3 (the other types inheriting the attribute).

SpicyPizza is a pizza with at least one spicy topping. Prior to defining the type, each topping has to be classified according to its spiciness. The SpicyPizza type can then define an additional requirement to have a hot topping. This hot topping will also satisfy the inherited hasTopping constraint.

Figure 9.5 shows the model focussed on the special pizza types. It stands out that both SpicyPizza and HighCaloriePizza are defined on O2 and not O1 as are the vegetarian and interesting pizza. The reason is that their definition takes place on a more concrete level of abstraction. Whether or not a pizza is spicy cannot be determined on the types of topping, because for every type of topping there can be spicy and non spicy variants. Only the instances carry the trait of being spicy or not.

Of course, each ToppingType could define a subtype SpicySeafoodToppingType, SpicyMeatToppingType etc. and there could be a common subtype SpicyToppingType of all those spicy subtypes. This would raise the need to classify each topping type to either have a spicy subtype or not, expressing it through a clabject and a generalization. The second alternative would be to model SpicyToppingType and a subtype of PizzaToppingType as a sibling of all the other topping types. A SpicyPizzaType could then have at least one of those toppings. In that approach, every topping instance would need to specify whether or not it is spicy through a second classification (besides the one pointing to the seafood, meat, ... type).

The reason the high calorie pizza is defined on O2 is slightly different. The commonality is that the definition of the domain concept cannot be completed on O1. The deriving question is: Can it be completed on O2? The answer is still not strictly "yes" or "no". One could argue that the calorie value of a pizza differs with each individual, depending for example on which brand of cheese is used by the manufacturing pizzeria. According to that line of argument, the level to specify that on would be O3. The opposing argument would assume that the calorie value of each individual pizza of one type is the same because their manufacturing process and ingredients are identical. Such a pizza would be manufactured in a factory. In that line of argument, the correct level would be O2. In each case, the definition of the criterion has to be one level above where it is specified. The multi-level modeling example takes the same viewpoint as the OWL tutorial in

assuming each individual has its own calorie value. So the Pizza clabject on O2 has a trait stating that each instance has to provide its calorie value, otherwise it won't be a pizza. The subtype HighCaloriePizza then sets the value and states that no instance may change that value. As the fixed value is not a single one but a range, a range of calorie values will be sufficient to satisfy the mutability constraint.

9.1.6 Reasoning

Reasoning on this ontology can both validate expressed classification and discover new ones from artefacts. MargharitaPizza will be classified as a VegetarianPizzaType as well as a CheesyPizzaType. As the current prototype tool defaults to the closed world assumption the classification will succeed. Because the entered information is considered to be complete, it is assumed that MargharitaPizza cannot have any toppings other than tomato and mozzarella.

The main use case for classification is of course classifying individuals as instances of one or more types. Figure 9.6 shows the individual pizza instances. All four pizzas are instances of MargharitaPizza, as all of them have a tomato and mozzarella topping. The individual toppings are explicitly classified as instances of one topping type. As they are all disjoint, the inference engine will not classify them as instances of another. If the types were not disjoint, one individual topping would be an instance of any topping, as the name cannot be processed by the tool. This principle separates the potential types for the individuals up to the generic type "Thing". It is therefore advisable to always have all the types in an inheritance hierarchy. SpicyPizza and HighCaloriePizza are not disjoint, because one pizza instance can be both, spicy and high calorie.

In the OWL tutorial the requirements for a type can be switched from necessary to sufficient. In the case of MargharitaPizza this would mean

Figure 9.6: The individual pizza instances

that an individual not only has to have exactly one tomato and one mozzarella topping but also no other topping. In multi-level modeling the way to express this would be to model a supertype of the two connections "hasTopping" with a multiplicity of 2..2, allowing no other toppings than those two. The consequence however would be that the other named pizzas could no longer be modelled as subtypes of margharita because it would be impossible for an instance of americanaHot to be an instance of margharita. In the OWL guide there is no explicit subtyping used, the superclasses are always inferred by the tool according to the actual properties. The mechanism of copying is used upon the creation of a named pizza to avoid redundant definition of object properties. The subsumption service can provide the information, but the current mode of operation would be to create generalization elements, thereby making the information explicit. If the alterations to the margharita type were made afterwards, it would result in an erroneous model that would have to be corrected by the user.

9.1.7 Advantages of Multi-Level Modeling

The previous section focussed on rebuilding the example from the OWL Tutorial (40) and explaining the differences that still exist with multi-level modeling. Up till now, it does not illustrate most of the novel features that PMLM provides:

Potencies help to state the intended number of ontological levels in the beginning of the process and can define the lifespan of model elements.

Property durability can do the same for properties that are suitable only up to a certain level of concreteness.

Type instantiation can create types or individuals from the template of another clabject, bringing the constructive use-case to the ontology.

Explicit classification somewhat contradicts the inference paradigm of OWL but can help to distinguish and classify the elements. In other words, it is information that can be used in the inference process as well.

Figure 9.7 shows the complete ontology. Notice that potencies vary inside one level. The reason is that the respective types do not all have isonyms on the next level. Many of the toppings are not used in the individual pizzas. So these toppings have potency zero on O1 because they do not have any instances. Seafood- and MeatToppingTypes are not used at all, so these types have potency one as none of their instances has a potency greater than zero. The other concepts all still endure until O3 which is not surprising as they were described in the example.

So the main contribution of multi-level modeling is a concise, comprehensive and semantically well founded concrete syntax which enriches the user experience when working with the data.

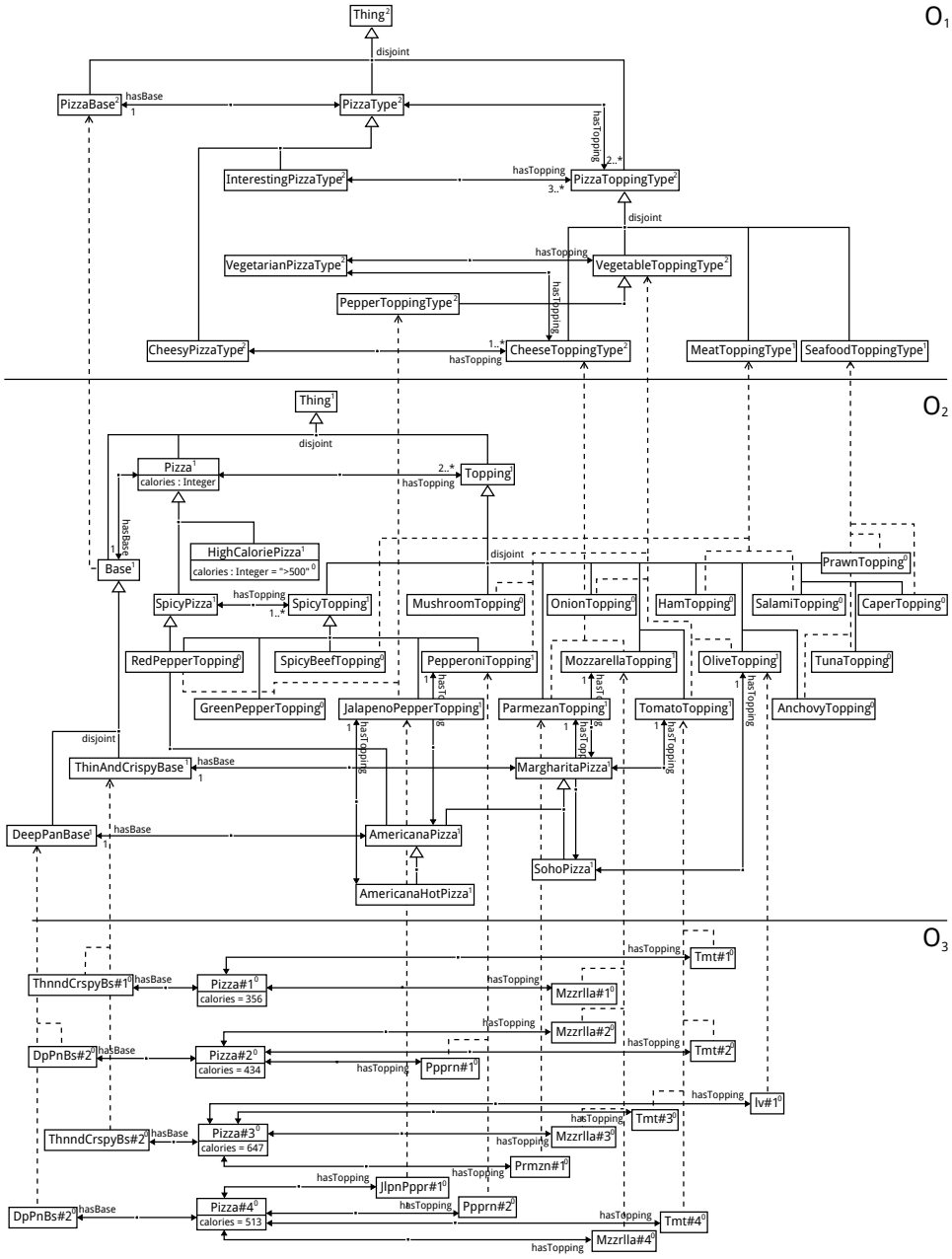


Figure 9.7: The complete pizza ontology.

9.2 The Royal & Loyal Example

The royal & loyal example is taken from the book by Warmer & Kleppe (73) and is the classic example for illustrating OCL expressions and constraints. The model is not so much focussed on inheritance and subsumption but on connections and model navigation. It has fewer generalizations and many more features. The total number of clabjects is lower but each clabject is described in more detail. The model from the book is shown in figure 9.8.

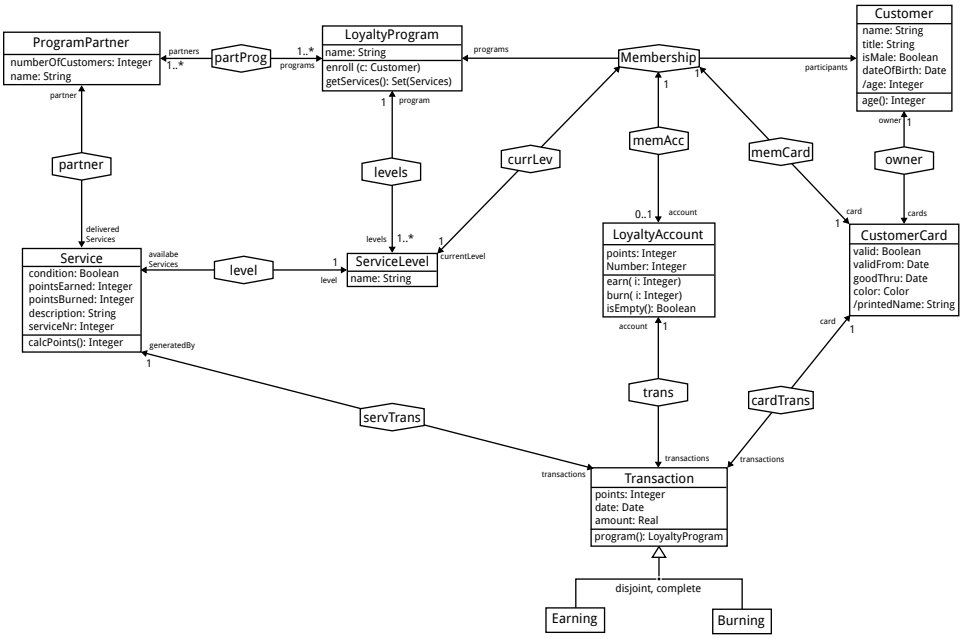


Figure 9.8: The royal & loyal model rendered in LML

The ontology in figure 9.8 is only one level (hence it can also be called model). In the example there are several small object diagrams which are then validated against OCL invariants or other constraints placed at the respective types. Most of the constraints cannot be modeled with PLM elements. They can be entered as OCL constraints with the respective clabjects, but that is not a contribution of multi-level modeling. PMLM *can*

contributes two things:

- Depending on the properties of the modeled domain more than two ontological levels can be introduced
- With the constructive user services defined on single clabjects the whole model can even be instantiatied.

The first step is to analyse the modeled domain with regards to involved concepts and their position in the stack of ontological levels. Afterwards, the leaf model can be instantiated.

9.2.1 Analysing the model

Before the domain is divided into separate ontological levels it is important to analyse the relationships between the clabjects. The interdependencies often form the basis for identifying parts that are inseparable. Even if there are no independent subparts of the model the analysis provides valuable domain insight.

9.2.1.1 Classification dependencies between the entities

Another question is the minimal set of instances that is needed so that any classification can be correct. Without connections, no entity need exist in order for another entity to be able to enter a classification. If connections exist, their multiplicity is interesting. If the lower multiplicity of each role in a connection δ is zero, the destinations can be types of clabjects without the presence of an instance of δ.

for $\psi : \psi.\mathsf{connection} = \delta$ **do**
  if $\psi.\mathsf{lower} > 0$ **then**
    return δ is necessary
return δ is not necessary

If all the lower multiplicities of the roles one clabject γ is the destination of are zero, it means that all the entities can be instantiated without an instance of γ being present.

for $\psi : \gamma = \psi.\mathsf{destination}$ **do**
  if $\psi.\mathsf{lower} > 0$ **then**
    return γ is necessary
return γ is not necessary

In the royal & loyal model, there is one unnecessary connection and three unnecessary entities. Transaction participates in three connections, but every time with multiplicity *. The subtypes Earning and Burning add only the complete partitioning of transactions, so they are unnecessary as well. Transactions are not required for the system to be well defined. They are the variables that operate the programs but every entity may exist without being linked to them.

The connection Membership has two roles, and both have multiplicity *. So neither of the participants (LoyaltyProgram and Customer) need a membership. The statement is that customers do not need to be a member of a LoyaltyProgram (it is not mandatory). Accordingly, a loyaltyProgram does not need any participants. In reality, while this might be quite frequent, a loyaltyProgram with no customer is not very useful. Interestingly enough, membership participates in connections itself. Each LoyaltyAccount needs a membership to exist and so does a CustomerCard. A customer cannot have an account without being a member of a program. So earning or burning points without being in a program is impossible[1]. Also, no customer can have a card without being in a program[2]. So, despite being unnecessary for the participants, the membership connection is crucial for other connections

[1] Interestingly, being a member without having an account is possible. As each transaction requires an account, it could be possible to become a member, but the account would be created upon the first transaction.

[2] Does a customer have to give back his/her card upon terminating his/her last membership? Is a check mechanism implemented in the program?

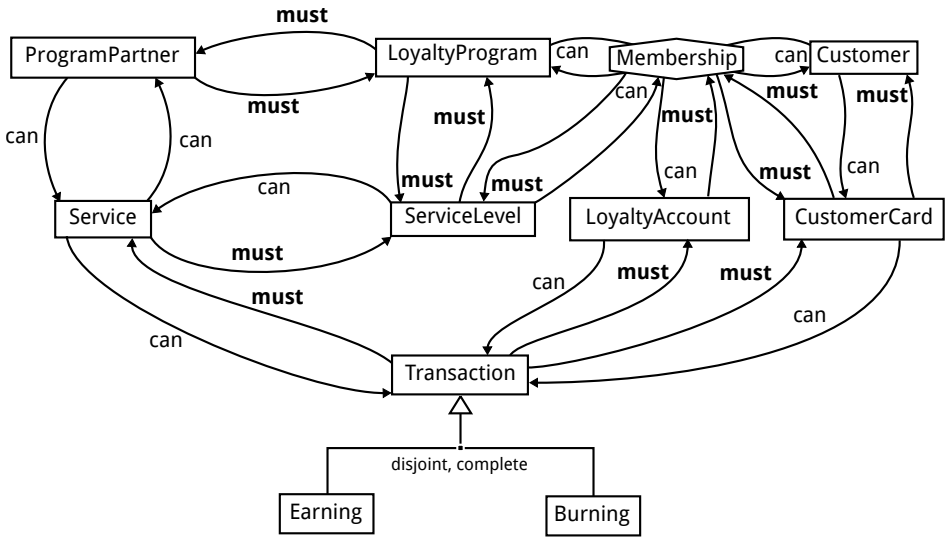


Figure 9.9: Royal & Loyal model with connections as classification dependencies

and therefore also entities. Formally, the connection check says membership is unnecessary, but the clabject definition overrules it.

9.2.1.2 Modeling classification dependencies

Figure 9.9 shows connections modeled in a slightly different way to show their consequence for correct classification. If a role has a lower multiplicity greater than zero, a "must have" dependency is established. If the lower multiplicity is zero, a "can have" dependency is drawn instead.

No other clabject must have a transaction. A Service can have a ProgramPartner, but a Service must have a ServiceLevel, a ServiceLevel must have a LoyaltyProgram and a LoyaltyProgram must have a ProgramPartner. As a consequence, there can never be a valid Service without there being a valid ProgramPartner. The requirement is that they are connected. Table 9.1 gives an overview of the dependencies in the whole model.

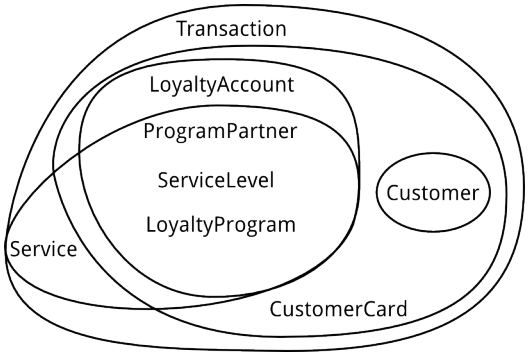


Figure 9.10: Royal & Loyal dependencies in set notation

Source	**Dependencies**
Transaction	Service, ServiceLevel, LoyaltyProgram, ProgramPartner, LoyaltyAccount, CustomerCard, Customer
ProgramPartner	LoyaltyProgram, ServiceLevel
LoyaltyProgram	ProgramPartner, ServiceLevel
ServiceLevel	LoyaltyProgram, ProgramPartner
Service	ServiceLevel, LoyaltyPrograM, ProgramPartner
Customer	
CustomerCard	Customer, ServiceLevel, LoyaltyProgram, ProgramPartner
LoyaltyAccount	ServiceLevel, LoyaltyProgram, ProgramPartner

Table 9.1: Classification dependencies of the royal & loyal model

Customer is the only entity without dependencies. A transaction depends on the whole model. ProgramPartner, LoyaltyProgram and ServiceLevel form a clique. LoyaltyAccount depends on this clique, but is not a part of it. With this information, the dependencies can also be rendered in set notation, as shown in figure 9.10.

9.2.2 Introducing Multiple Ontological levels

Based on the analysis of the dependencies of the clabjects, the model can be separated into two ontological levels. It seems odd that none of the entities needs transactions, and they in turn depend on everything else. Customers are only needed by CustomerCards, but the removal of customers would remove memberships as well, and they have many dependencies.

This can be resolved by considering different possible forms of acceptable clabjects. A ProgramPartner could be a company or concern, a LoyaltyProgram some program they choose to offer and a Service a building block to implement it. Can there be instances of these as well? One LoyaltyProgram may be offered by a branch of the company (a ProgramPartner instance) and the concrete service in operation would be an instance of the service (instance) offered by the company.

At which level do transactions and customers come into play? An individual (having a name and a birthday) clearly is on the same level as the branch of the company offering the concrete service. So the type describing that individual must be one level above, together with the company. Concrete transactions also take place between individual customers and concrete services offered by one branch of the offering company.

Figure 9.11 shows the two level version of the original model. The choices giving rise to this ontology have several consequences: The membership connection can only be introduced at O1, not O0. The enroll operation of a LoyaltyProgram can also only be introduced at O1. That makes the instances in O1 hyponyms of their types. If the objective is to create isonyms, the concept of a customer (and customerCard, transactions and accounts) have to be abstracted to O0.

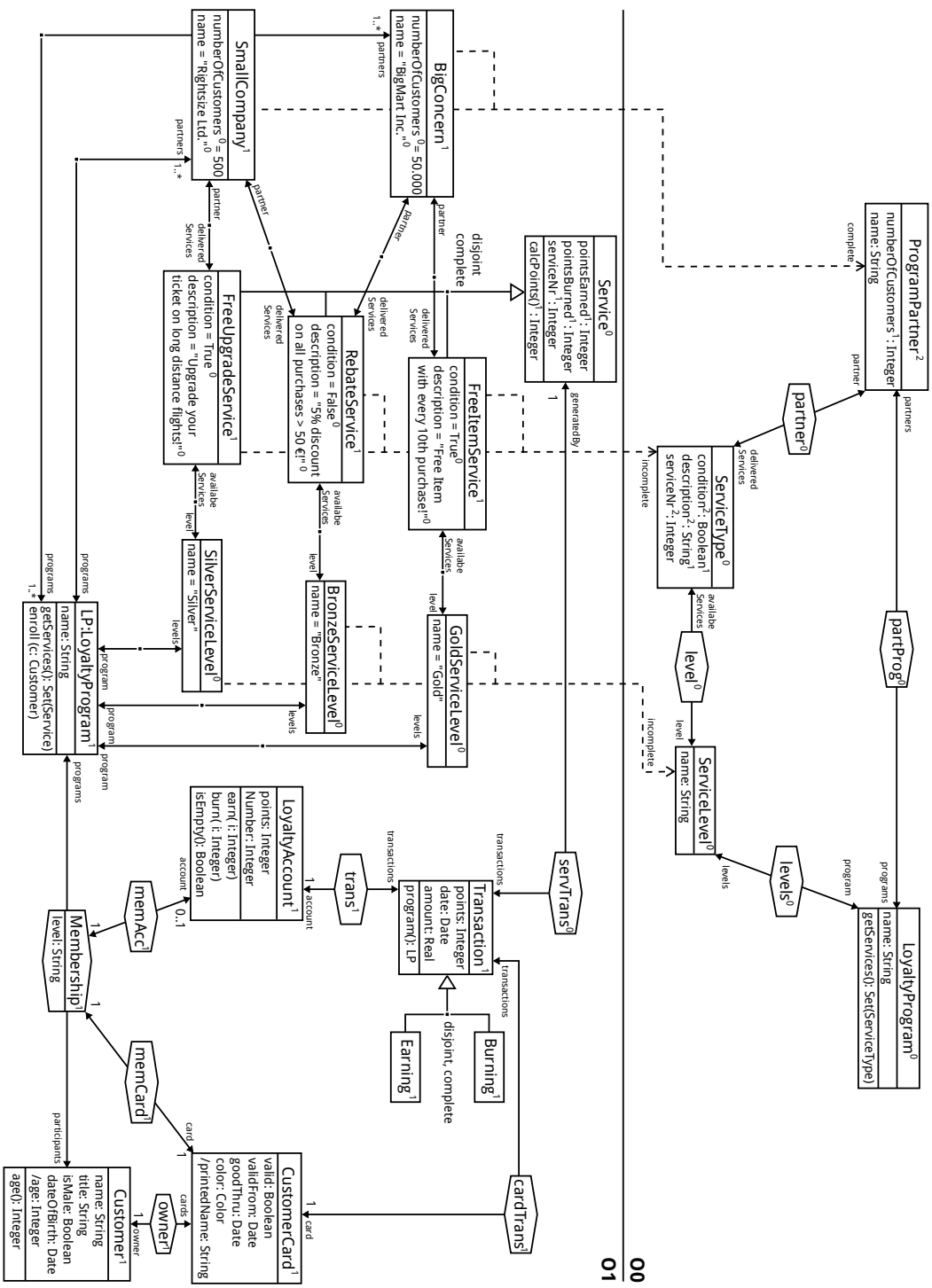


Figure 9.11: Royal & Loyal initial model split in two levels

9.2.2.1 Potencies

At first sight the potencies in figure 9.11 seem odd. All the connections on O0 have potency zero. In fact, the only clabject with potency two (the value one would expect at O0) is ProgramPartner. The defining concept for potency are the type properties and the resulting partition of the instances into isonyms and hyponyms. A ServiceType isonym can only have three features: condition, description and serviceNr. The services FreeItemService, RebateService and FreeUpgradeService have six — the two they define themselves and the four they inherit from Service. Without the inheritance from Service, they would neither have the features required to be an instance of ServiceType nor would they define the required point features for O2. So they cannot be isonyms of ServiceType and hence the potency of ServiceType is zero. This immediately explains why the potencies of the connections ServiceType takes part in cannot have a potency other than zero. A connection potency can never be higher than the minimum of the potencies of the participating clabjects. For the other participants (ProgramPartner and ServiceLevel in this case) it means that isonyms do not need to redefine the connection as its existence on the next level is not mandatory[1]. LP cannot be an isonym of LoyaltyProgram because it defines a connection not required by the type: the newly introduced concept of Membership. The ProgramPartner instances on the other hand define only connections where there are types (not complete ones) on O0. As BicConcern and SmallCompany do not define any other connections, they are isonyms of ProgramPartner. PartProg can only have potency zero because of LoyaltyProgram.

The potency definition on O0 leads to the insight that the domain setup is very complicated and incomplete. Since all the clabjects on O1 except the ServiceLevel instances (which have potency zero on purpose) have potency

[1] That is the reason why there are no multiplicities on O0

1, the domain of O2 is very tightly constrained. The multiplicities of the connections on O1 are the same as in the original example and the object diagrams used in the example represent valid snippets of a consistently classified model for O1. The resulting model looks much like the original one (figure 9.8). If some of the choices were made differently there could be obvious potencies but the link to the original would be lost.

9.2.3 Instantiating O1

The instantiation of O1 follows the principle of creating the smallest possible model that is a consistent classification of O1. Only if the created instantiations are consistent, do the potencies in O1 become valid and the whole ontology is complete. By instantiating O1 into O2, the mode can then also switch as the ontology is not complete at the moment, only consistent. The transition into exploratory mode requires complete potencies and therefore isonyms on O2.

9.2.3.1 Creating Local Offspring

Local offspring have to be created for all the clabjects with potency one in O1. These are:

BigConcern, SmallCompany, FreeItemService, RebateService, FreeUpgradeService, LP, Transaction, Burning, Earning, LoyaltyAccount, Customer, CustomerCard, Membership, memAcc, memCard, cardTrans, owner, trans, partner and partProg.

Initially, there will only be one offspring for each connection. The missing required connections to resolve the multiplicities have to be added later on. It is apparent that servTrans does not obtain a local offspring in this approach because its potency is zero. Its potency has to be zero as the participant Service is also of potency zero. That does, of course, not mean that there cannot be any instances on O2, just that the engine will not build them on its own as they are not strictly required.

Figure 9.12 shows the newly created offspring clabjects on O2. The mutability of the name of the program partner does not need to be shown any more as it is now the default value as it equals the attributes durability (and the clabjects potency). Many of the clabjects are basically meaningless. That is because their properties are still missing and also the traits with domain relevance (like name) have not yet been added by the user. The roles are not present and the attributes have the standard value of the datatype unless the value is fixed from the type.

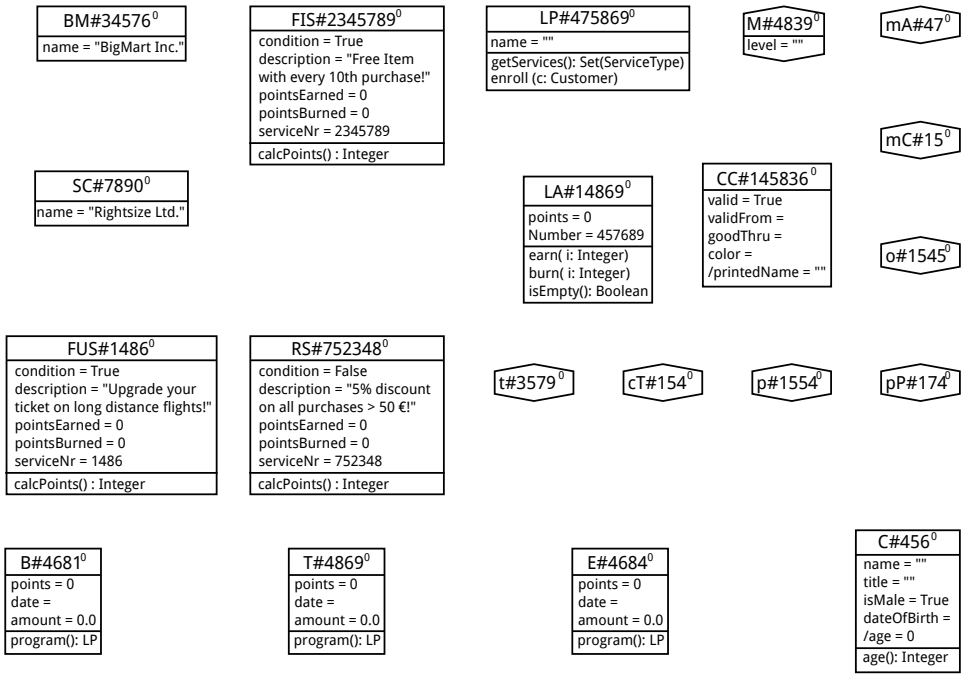


Figure 9.12: The created offspring of the royal & loyal ontology

9.2.3.2 Connecting the Participants

An engine's first first attempt to connect the participants will only create the roles for the present connections. The user can already guess from the look of O1 that some missing connections still need to be added to complete the ontology.

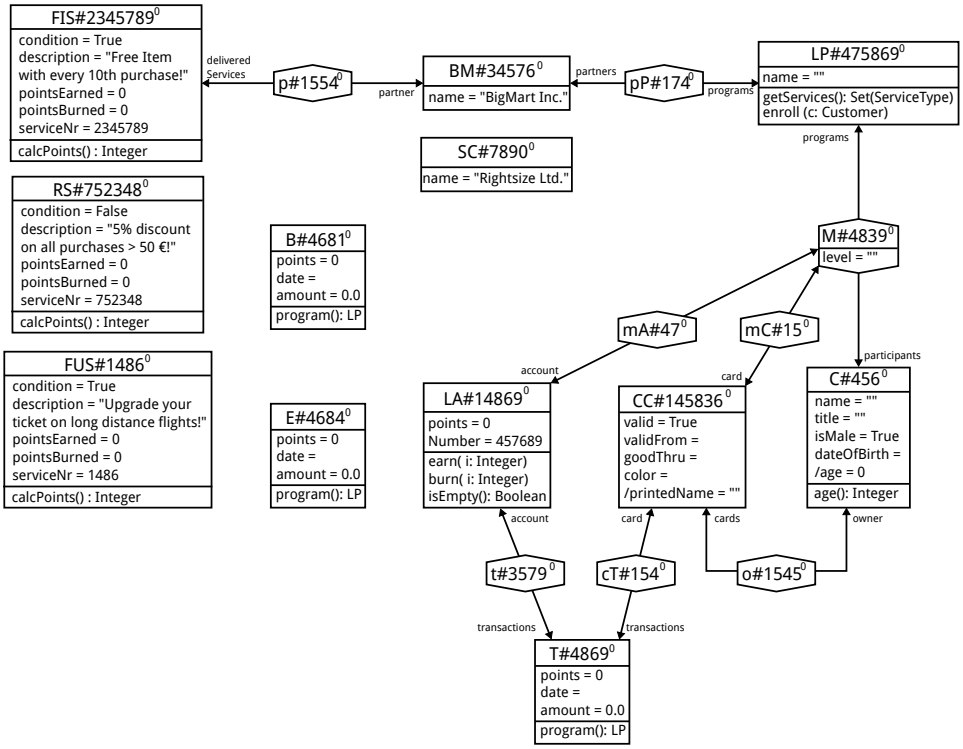


Figure 9.13: The connected offspring of the royal & loyal ontology

As the link between blueprint and the offspring is maintained it is not a problem to find the participating clabject and roleName for each role of O1. Figure 9.13 shows the ontology after the creation of the roles. The transaction type has been chosen as the participant, despite the fact that only earning and burning transactions will every be valid. The transaction type itself would be abstract, but the potency is correct as every earning and/or burning isonym will always be an isonym of transaction as well. If the model is created from scratch it is most likely that the loyalty program will be empty in the sense that no transactions have taken place yet. The existence of a loyalty account and its connection to a membership is mandatory given its potency.

9.2.3.3 Checking the multiplicities

After the roles have been created[1] the number of missing connections can be determined by checking the multiplicity of the connections in O1. Once the multiplicity is complete, the ontology will be as well. The resulting model will of course be a small one. If all the lower multiplicities in a connection in O1 are zero, there is no need for any instance (except for the one required by the potency), so the check for ontology completeness will not require it. The user is of course free to expand any model as needed to best capture the domain in hand. Figure 9.14 shows the complete model.

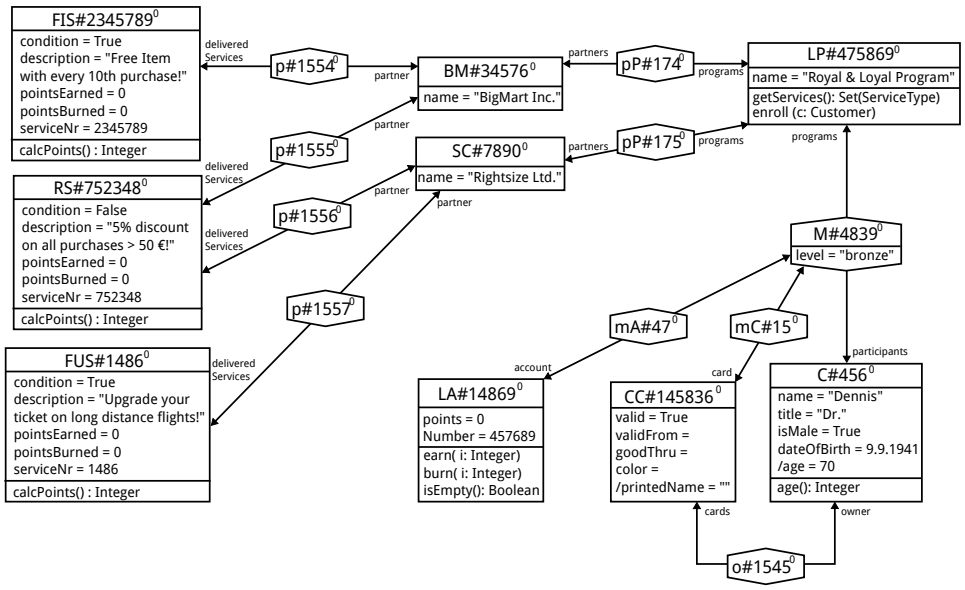


Figure 9.14: The complete model O2 of the royal & loyal ontology

[1] and there is no need for correlations since all the clabjects have potency zero

Chapter 10

Related Work

Now that the main contributions of the thesis have been presented, analysed and shown in practice on two case studies, it is now time to compare it against other work in this field. Wherever possible, the differences between the approaches is explained in the terms established earlier and the possibilities for collaboration and cross fertilization are discussed. The presented approaches all deal with multiple levels of modeling in any aspect. However, there have been numerous modeling-based approaches to enhancing the state of the art that shall not go unmentioned. Model-based energy testing(75) establishes static analysis of the energy consumption of software components.

10.1 Metamodel Semantics

Prior to the advent of multi-level modeling, there has already been research on metamodel semantics. When analysing the semantics of any language, there needs to be both a language definition and instances of that language definition to validate the semantics against. So there needs to be at least two levels of abstraction for semantic discussion to take place. The OMG four layer architecture (55) provides even three of these instantiation steps

and has been subject to extensive semantic research. Apart from the concrete metamodel semantics presented in the following paragraphs, reference attribute grammars (RAG(38)) provide a promising approach to metamodel semantics(25).

(61) analyses the MOF semantics through constructive type theory while staying agnostic to the actual instances of the MOF used. So the work is applicable but not limited to the UML metamodel as an instance of the MOF. Comparable work (20) uses algebra to define semantics and also lower level languages like UML metamodels (64) are covered. These approaches follow the same principle. A formally defined system is used to represent the MOF and instances of the MOF are then expressed with the formal system so that their adherence to the MOF's constraints can be formally ensured.

Although a new formalism of semantics for potency based multi-level modeling might seem redundant to the already presented work, as the representation of a metamodel in a formal system and the classification of instances is already covered there is one important difference.

In the OMG four layer hierarchy (55), the classification applied between MOF and the UML metamodel is linguistic and not ontological. That argument is easy to state, but not so easy to prove because there is not yet a formal distinction between ontological and linguistic classification. The MOF defines concepts such as classifier, attribute, datatype and association. The concepts defined in the UML metamodel are basically the same: class, attribute and association. Apart from the names of the types being the same, the definitions of the concepts are also very similar. In fact, since UML 2.0 the MOF is supposed to be a subset of the UML metamodel defintion. So the UML metamodel is not an ontological instance in the sense that its building blocks are used to construct information about a target domain, but it is a linguistic specialization in the sense that the defined concepts are enriched, reused and extended.

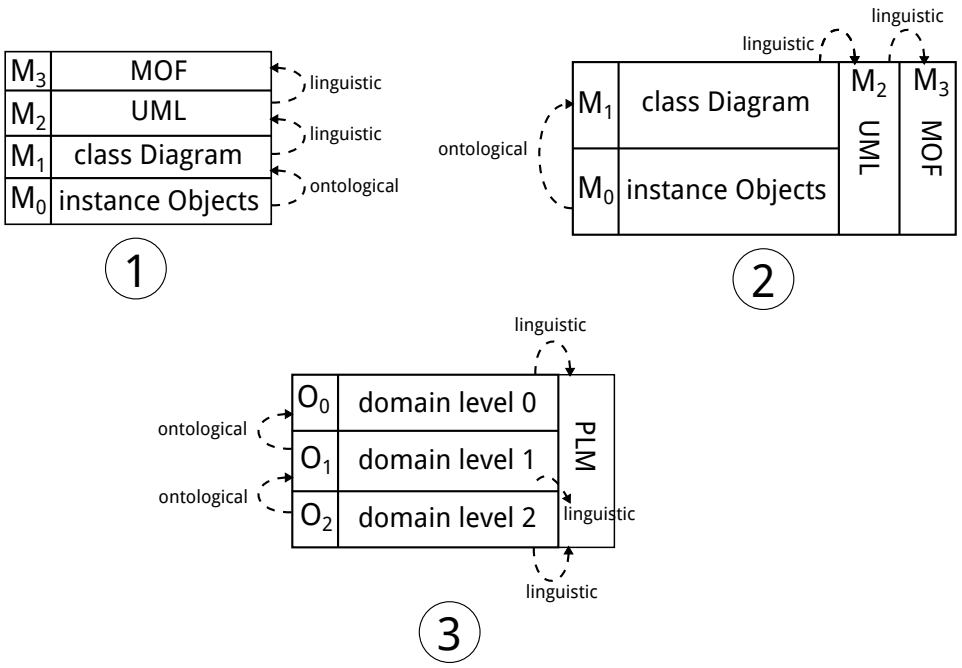


Figure 10.1: Different kinds of classification in the OMG four layer hierarchy

The classification between the UML class diagram and the UML Metamodel is again linguistic. This is the classification step equivalent to the creation of PLM instances. Only the classification between the UML class diagram and the instances of it is ontological (compare (1) in figure 10.1).

If we arrange the visualization schema of linguistic classification horizontally and ontological classification vertically, we get (2) in figure 10.1. The end user (the person using a modeling tool) will not be aware of the linguistic levels other than the one he/she uses to create the model elements from. So with multiple ontological levels of classification being present, there is no need to have multiple linguistic levels as well, as the user will only ever use one. In fact the goal is to ensure that the single linguistic level is sufficient to represent all the ontological ones and the user does not need to resort to different languages depending on the ontological level of abstraction. For this reason, there is only one linguistic metamodel in potency based multi-

level modeling, the PLM. The evolution of different model layers is shown in figure 10.1.

If the semantics of multiple linguistic levels are viewed from the perspective of a person using a modeling tool, these semantics are hidden in the implementation of the tool. If the implementation of the metamodel is correct, the produced models will be valid according to the semantics of the (UML) metamodel. If the tool violates the semantics of the metamodel, the resulting models will probably not be processable by other tools using the same standard. For the end user, such semantics do not provide any benefit, but are a necessity for usable tools. The semantics for ontological classification on the other hand can give immediate feedback on the information the user expressed about the modeled domain. If inconsistencies arise in such situations, the user can process the feedback and refine the model (or the view on the problem domain).

10.2 MetaDepth

MetaDepth(47) is a multi-level modeling kernel building on the foundations set out by Atkinson and Kühne (9, 12) as well as the early PLM publications (15). The authors adopt most of the basic concepts and realize them in a state-of-the-art modelling framework.

10.2.1 The main characteristics of MetaDepth

Constraints MetaDepth features linguistic support for constraints using the Epsilon Languages (44).

derived Attributes The metamodel has a subclass of Field for derived attributes whose value is not set by the user but derived via a computed expression.

transactions the core API calls are recorded in an event list and by persisting this list not only are the results reproducible, the steps can be re/un-done and grouped into transactions.

MetaDepth features two modeling modes, strict and extensible, which control whether or not the definition of a type can be extended by the ontological instances.

Code Generation By default the MetaDepth kernel runs in interpreted mode as intended by the model driven development paradigm, but in case more efficient but fixed Java code is needed the kernel is able to produce it.

MetaDepth has three modes for entering data to the model kernel: a Java API, a command shell and a textual syntax. The latter is a particularly useful way of representing small (multi-level) models. A graphical concrete syntax is not one of the primary design goals for the MetaDepth kernel.

```
Model Store@2 {
 Node ProductType {
  VAT@1          : double = 7.5;
  price          : double = 10;
  discount       : double = 0;
  minVAT@1       : $self.VAT>0$
  minPrice@2     : $self.price>0$
 }
}
Store Library {
 ProductType Book { VAT = 7; }
}
Library MyLibrary {
 Book mobyDick { price = 10; }
}
```

Listing 1: MetaDepth Model stack example

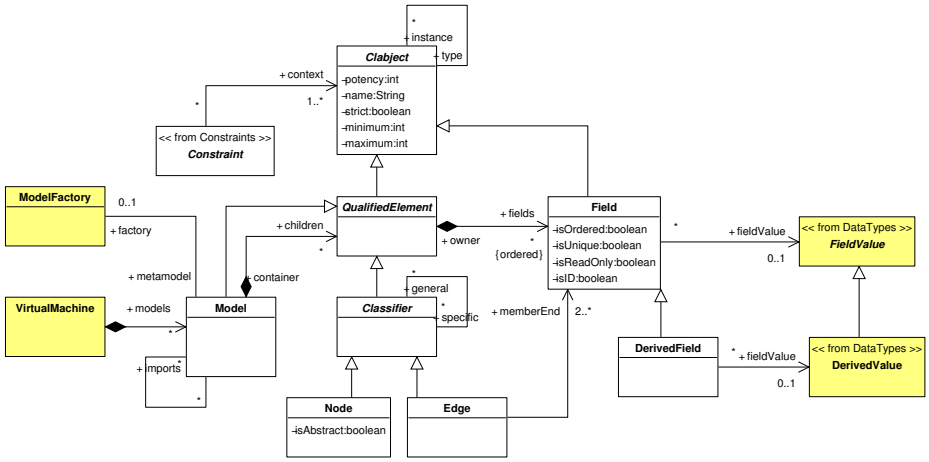


Figure 10.2: The MetaDepth (partial) metamodel.(47, p.6)

10.2.2 What MetaDepth and the PLM have in common

Model MetaDepth uses the same notion of Model (the logical set of all the elements of one ontological level) as the PLM.

modeling modes The strict and extensible modeling modes proposed by MetaDepth are similar to constructive and exploratory modeling modes of PLM. In strict mode, there can only by isonyms, never hyponyms, and the reason for making a model extensible is the same, enabling hyponymic instances.

Undefined/Unlimited Potencies MetaDepth features unlimited potencies like PLM. The concrete syntax * is also the same.

default values The default values for feature potency (and potency in general) is the same as well as the understanding about the visual impact of a default value: They need only be shown if the value differs from the default.

Read-only fields and the accessibility of a fields implicitly addresses the issue of value potency (mutability). If a constraint includes several

ontological levels and uses fields from those levels, the field from the higher levels is treated as a "field of the type" (a static field) which is the same intention as assigning default values to attributes in PLM. Together with read-only, a field could be read-only and have potency > 0, which would be equivalent to an attribute with durability 1 and mutability 0.

The connections between nodes are first-class citizens of the model. They are clabjects, they can hold fields and participate in other connections themselves.

n-ary associations Associations can have an unlimited number of participants, just like PLM connections.

```
while (Transition.allInstances()->exists(t | t.enabled
    () and t.fire())) {}
operation Transition enabled(): Boolean {
 return self.ArctPT->forAll(arc|arc.inPlaces.tokens>=
     arc.weight);
}
operation Transition fire(): Boolean {
 for (arc in self.ArcPT)
  arc.inPlaces.tokens := arc.inPlaces.tokens - arc.
      weight;
 for (arc in self.ArcTP)
  arc.outPlaces.tokens := arc.outPlaces.tokens + arc.
      weight;
 return true;
}
```

Listing 2: Epsilon Object Language simulator for Petri nets

10.2.3 What Distinguishes MetaDepth and the PLM

Clabject multiplicity MetaDepth clabjects define multiplicity bounds to control the number of instances that can exist in the instance model.

Such a trait is not present in the PLM and would require further research as the PLM differentiates between different kinds of instances which are not fixed through the lifecycle of the model.

Model nesting MetaDepth models can import other models and therefore effectively nest them inside. Currently no such interaction feature between models of similar ontological levels is implemented in PLM.

Model potency MetaDepth models are clabjects themselves. As such, they have a potency, can hold fields, define their mode and multiplicity. The potency of a model is utilized in a model factory to instantiate the whole model. The modeling mode decides whether or not an instance model can add new clabjects to it. Some of the implied features are realized in the PLM in a different manner: The modeling mode is globally derived. Fore example, if a model contains * potency clabjects, it cannot be exploratory. In the current design the PLM authors do not see the need for the level of detail a mode per model offers. The potency of a model defines whether or not the model as a whole can be instantiated in one batch. While the question remains whether a contained clabject can have a higher potency than the surrounding model (and what the semantics of such a setup would be), the PLM authors again see no benefit at this level of detail as the batch instantiation of a model is the sequential instantiation of the contained clabjects and each Instantiatability is defined by the individual potency. The situation is the same for model fields.

Ontology In MetaDepth, the VirtualMachine is the top level container for model elements. It is a singleton element containing all the models. In the PLM one stack of ontological models is contained in one Ontology. There may be more Ontologies in one system and interoperability is defined on the ontology level rather than on the model level.

Exploratory mode In MetaDepth the extensible modeling mode allows instances to define more properties than required. Nevertheless the basic paradigm of modeling only in descending potency remains. True exploratory modeling also covers the use case of deriving types from instances or examining given levels of elements without any prior potency constraints.

association traits MetaDepth utilizes fields to define association ends, resulting in a very concise linguistic definition because the information about participation is stored in the field of the participating clabject. PLM chose to follow a different path and stores the linguistic information in the connection itself. The main difference however is that MetaDepth chooses the same fields as for data storage. So the connection participation does not have a separate roleName or a configurable navigability. Multiplicity is implemented naturally through the fields multiplicity.

Fields are Clabjects In MetaDepth, clabject is an element very high up in the meta model inheritance tree. In PLM the only Clabjects are model nodes and edges, in MetaDepth basically everything except a constraint is a clabject. The main consequence of this difference is that the element is a clabject in MetaDepth, but not in the PLM. For fields, the difference sounds bigger than it is. In the PLM a feature is not a clabject because it a) cannot contain other artefacts and b) cannot have instances or types on its own. In MetaDepth recursive containment of artefacts is not possible and the concept of instantiation is defined in a lightweight way so the implications for fields having instances are not as far reaching as they would be in the PLM.

linguistic support for methods In the PLM, methods are features in the same way as attributes/fields are. In MetaDepth, methods are defined

through an expression in the action language pinning them to a context. So in the PLM methods are artefacts used to indicate that a model element has to offer an operation and in MetaDepth operations are attached to an element indicating the ability to perform a computation. The difference is subtle and perhaps originates from the absence of an expression language like Epsilon in the early design phase of the PLM.

linguistic support for correlations Correlations like classification or generalization are realized in MetaDepth in a very lightweight way: A linguistic link in the metamodel. The information itself is not represented by a distinct element in its own right as with PLM classifications or generalizations. The PLM elements can hold information themselves (disjointness, completeness etc.). The motivation for embracing correlations in the metamodel (PLM also features SetRelationships which are not present in MetaDepth) is to enable reasoning on those elements. Checking whether or not a generalization relationship exists is in theory also possible in MetaDepth, but not to the same extent as in the PLM. The information contained in the artefacts of the model may be discovered as well (after all, it is there), but it can not be shown in the model in the form of computed elements as in the PLM.

linguistic support for rendering information The use case of MetaDepth does not cover the modeling of the rendering of an element, only the artefacts the element represents. The PLM defines linguistic metamodel elements for rendering that control how the elements are rendered in a graphical way. This would be achievable in MetaDepth but is not yet built into the metamodel.

10.2.4 Conclusion

With regard to the research goals it is safe to say that MetaDepth does not focus on Research Goal 2 and Research Goal 3. As such, it also cannot fulfil these goals or any of the related hypotheses. The main goal of MetaDepth clearly is Research Goal 4. Research Goal 1 is not in scope for MetaDepth either as it does not provide a visual editor. In terms of the observed weaknesses, MetaDepth clearly solves the two level (Fundamental Weakness 5) and linear modeling (Fundamental Weakness 4) weaknesses, but fails to overcome any of the others (fragmentation, assumptions and concrete syntax). Nevertheless the textual notation proposed in MetaDepth is very efficient and human readable.

10.3 OMME

The **O**pen **M**eta **M**odeling **E**nvironment (67, 68, 69, 70, 71) was developed in the group of Jablonski at the university of Bayreuth. It aims to provide a foundation for integrating different languages within the same model to combine their strengths and use them alongside each other in the development process. The theoretical foundations are the same ones as for the PLM: The orthogonal classification architecture, clabjects, deep instantiation and the separation of linguistic from domain specific classification. The implementation of the tool also follows the same principle as the Multi-Level Modeling And Ontology Engineering Environment (Melanie) (33): eclipse plugins based on Ecore. The tool will[1] be presented by the group at the world's largest computer fair, CeBIT in 2012.

10.3.1 Metamodel

Just like the PLM, OMME features an orthogonal metamodel called the **L**inguistic **M**eta **M**odel (LMM, figure 10.3). The design goals for defining

[1] at the time of writing

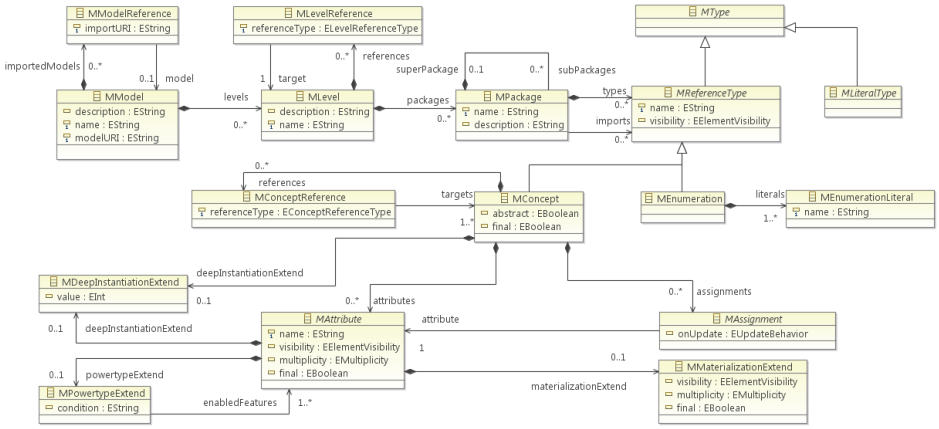


Figure 10.3: The **L**inguistic **M**eta **M**odel (LMM) of the **O**pen **M**eta **M**odeling **E**nvironment, taken from (70)

the LMM were not only to support advanced modeling concepts but to maintain flexibility in terms of a special modeling paradigm. The structure of the produced models shall not be constrained in any unnecessary way, including a strict semantic for classification (as in the PLM), making way for non-linear model stacks and arbitrary references between models. The top level (in the sense that there is no level above it) breaks with the self defining nature of the OMG MOF (55) and the names used in the metamodel are kept intentionally neutral, trying to avoid any bias towards any domain, even computer science.

The central part of the metamodel is the concept. Concepts hold attributes, assignments and references and are contained in Packages which are contained in Levels which are contained in models. Levels can reference other levels with three reference types: *instanceOf* for classification, *references* for a coupling that allows generalization between the levels and *alignedWith* for a link with little semantics. Concepts can reference each other with four types: *instanceOf* for classification, *extends* for generalization, *partitions* for powertype references and *concreteUseOf* for specializing

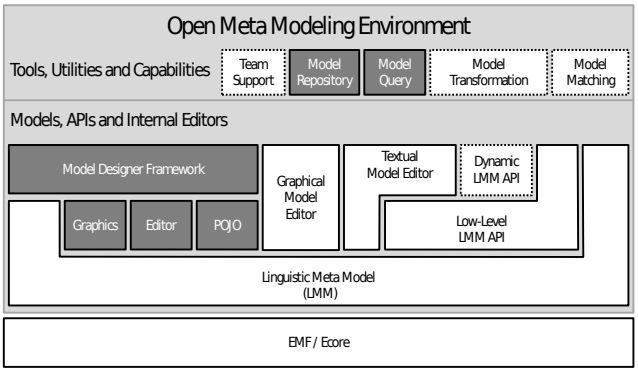


Figure 10.4: The implementation architecture of OMME, taken from (70). Only the white boxes with solid lines are operational at the time of writing.

the instance facet. The powertype pattern is a first class citizen of the meta-model and is defined on attributes. Other extensions like deep classification are defined in the Extensions package with a *MDeepInstantiationExtend*, which basically is a container for a potency value. The deep instantiation can be defined on attributes and concepts, so attributes can have a potency as well. Connections are missing from the meta model and are modeled as attributes where the value is another concept. Thus connections are not first class citizens in the sense that they can not be specialized and cannot have attributes.

The implementation structure (shown in figure 10.4) of OMME builds on EMF (24) and the connected frameworks for graphical editor creation (GMF(30), GEF(29) and Xtext(31)). The meta model is implemented as an Ecore model and the other services are built on that foundation. An LMM API provides access to the LMM Elements of all levels. The models can be edited in both a textual editor and a graphical one. The appearance in the graphical editor can be customized with either predefined shapes or custom java implementations. If these definitions became a part of the model itself (and could at least be configured through the model), the mechanism would represent a step towards the same goal as the LML in providing the basis for

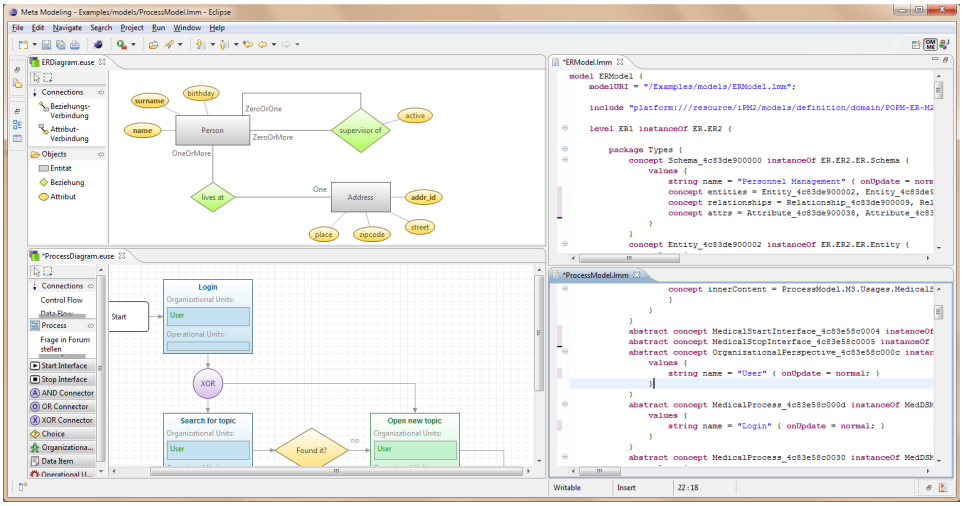


Figure 10.5: A screenshot of OMME, taken from (69)

defining domain specific languages inside the domain model. OMME further provides a SQL like query language for accessing their model repository. Melanie uses the EMF OCL implementation to access and navigate through the model graph, both in the editor via a context console and also through the model API in java code. Figure 10.5 shows a screenshot from the editor.

10.3.2 OMME Approach

The OMME approach builds on exactly the same fundamental insights and technology as the work presented in this thesis and the resulting tool Melanie. OMME tries to stay neutral in the naming of the elements and the implementation also tries to hide the names in favour of the domain specific ones. In that sense the LMM is not promoted as a common vehicle for different model stakeholders to fall back on for communication. The PLM is meant to do so. We believe that the Tower of Babel problem can be solved by providing a joint language for all of the stakeholders to communicate with. More importantly, the PLM is intended to be the common ground for both domain and technical experts to express their knowledge in.

The levels in OMME do not need to be in a linear ordering. In PLM, the levels are defined to correspond to the linear order of domain abstraction. In other words, the PLM levels are defined implicitly through the level traits of the clabjects (and therefore their level of domain abstraction), and the OMME levels are user defined containers for model content[1]. This allows a few constructs that are prohibited in PLM, such as generalization across level boundaries, clabjects of the same level in different models and classification within one model. In OMME, powertypes are a first-class citizen of the model and are explicitly supported through a metamodel element. For the development of the PLM, powertypes were a motivation because they are a prime example of the awkward workarounds the UML resorts in order to capture deep classification. So the PLM does not try to mimic or support powertypes, it aims to provide mechanisms to express what powertypes were invented for, but not as an ad hoc addon but as a fundamental design element. With multiple levels of classification, a powertype can simply be modeled using classification and generalization. An attribute and its value assignment are separated in OMME into different metamodel elements to mimic the type and instance facet of the owning concept. In the PLM, the affiliation of an attribute to either the type or the instance facet is determined by its durability (in OMME DeepInstantiationExtend.value). Furthermore, the type and instance facet are not exclusive. An attribute can belong to both and if it does, it does so as the triple (name,datatype,value). For the type facet, the value acts as a default value or in case of lowered mutability even as a type constraint for the instances. Also, in OMME the datatypes are contained within the metamodel to limit the possible "types" of attributes. While this proves beneficial for implementation on the eclipse platform it limits the datatypes the user can apply. For example, to represent a date, in OMME there needs to be a concept for it and this concept has

[1]It is difficult to not mix up terminology, as an OMME level is a PLM model, and a OMME model is a PLM ontology

to be referenced. The impact is not so big, as references between concepts and between literals are realized in the same way, but still the user is left with the task of providing the datatypes suitable for the domain. Melanie makes use of the dynamic evaluation features of the expression languages defined for the eclipse and EMF environment. On the downside, the static type checking of attributes becomes harder, but the user is given the extreme power to not only use all of the supported datatypes, but also to provide expressions as values which then evaluate to a valid value.

The definition of the LMM is lightweight with respect to correlations. Generalization relationships can be modeled as one kind of ConceptReference, but these references do not hold any traits and because of their setup can only ever connect one concept to n others. This combined with the fact that generalization are specified as a reference from the subtype to the supertype, makes it impossible to model one element that indicates that one concept is the supertype of several others. It is only possible to model several elements that state that one concept is the supertype of one other. The subtle difference is that concepts such as disjointness or completeness cannot be modeled. Intersection can be modeled (from the setup of the metamodel) but is never mentioned in the literature. Classification kinds are also not present.

The potency values (called DeepInstantiationExtend values) cannot be *. The lack of * potencies forces the user to always fix the number of classifications when using a concept. Even with the loose coupling of levels, instantiation of a potency zero concept or attribute is not possible in OMME (since this is the very meaning of deep instantiation). So although the containment concepts are designed with maximum flexibility in mind, potencies are strict.

The environment (again, at the time of writing) does not provide any semantics or formal definitions of classification. Thus, the modeling mode is ultimately limited to constructive modeling. Exploratory questions cannot be

answered. The SQL like model query language and several implementation blocks in the OMME architecture 10.4 indicate that exploratory questions are in the scope of the framework (Model matching, Model transformation), but as long as fundamental pieces of information can only be given by user input and not challenged, the provision of services present in the ontology world of modeling will be difficult.

10.3.3 Conclusion

The only research goal that is clearly achieved by OMME is Research Goal 4. While Research Goal 1 is one of the starting points as well, the reduction of accidental complexity has not really been acheived. The separation of attribute names and values, the explicit datatypes, the various ways to group model elements and the strictly binary generalizations mean that a lot more elements have to be created than are actually needed.

In terms of the observed weaknesses, OMME tackles Fundamental Weakness 3 even in a domain specific manner. Fundamental Weakness 4 and Fundamental Weakness 5 are clearly solved by OMME, but Fundamental Weakness 1 and Fundamental Weakness 2 are not covered.

10.4 OMEGA

The **O**ntological **M**etamodel **E**xtension for **G**enerative **A**rchitectures (OMEGA) (34) differs from the approach proposed in this thesis in two main ways:

1. It is realized as en extension to the MOF(55)
2. Its main purpose is code generation

The motivation to chose the MOF as a basis for extension seems legitimate as the MOF has proven effective in practice as a meta object facility. For PLM, MelAniE and the LML, the MOF is not suitable as it embodies many

of the shortcomings which are the motivation for the design choices made. A good example is the distinction of ontological levels in the usage of linguistical elements. The instances of attributes are slots. The PLM treats all levels uniformly in the sense that the linguistical elements used are all the same. Obviously that was not the goal for OMEGA, as the concept is even extended with the concept of a meta attribute, extending the possible linguistic meta elements to three. "Attributes" in intermediate levels have to be realized by two linguistic elements, one attribute as the instance of a meta attribute and one meta attribute as the type of the next attribute (or slot). Figure 10.6 shows the portion of the metamodel that is new or changed compared to the original MOF. The concept of having explicit linguistic types for both the type and the instance facet of a concept is applied also to Association, AssociationEnd and Class. This is perhaps the most fundamental difference to the PLM as one fundamental building block of the PLM is to treat ontological levels uniformly.

The second main difference shows that the purpose for creating the frameworks is totally different. As OMEGA is tailored towards the generation of source code, exploratory questions are totally out of scope. It provides valuable additions for creating source code which can then be enhanced. MelAniE, in contrast, tries to execute a model directly in the sense that the running system is a part of the model.

Potency is not realized as an integer in OMEGA but as a boolean attribute *canSpawnMeta* which determines whether or not the instances can be again MetaElements or must be instances. In that way, the sole purpose of potency is missed. Potency is meant to control the instances of the instances at the current level. With a boolean attribute, the choice has to be remade at every level.

Strict Metamodeling The inital definition "every model element must be an instance of exactly one element from exactly one level above" has

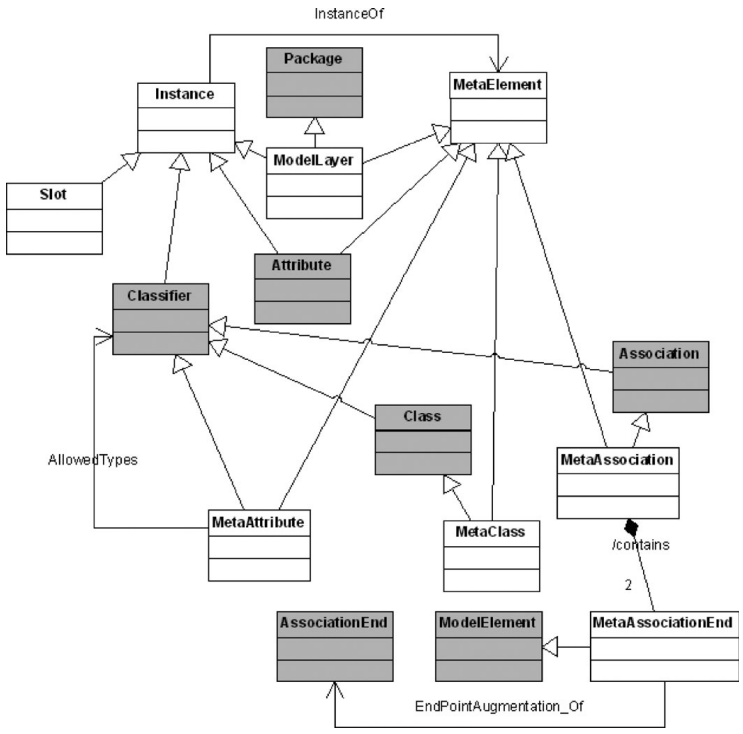


Figure 10.6: The OMEGA meta model overview. The grey classes have either been changed or are supertypes of newly introduced classes (shown in white), taken from (34)

been relaxed by OMEGA to "every model element must be an instance of exactly one element from a higher level". This definition still only allows exactly one type, but that type does not need to be on the level immediately above but can be at a higher level, allowing classifications to cross more than one level boundary. This relaxes the definition of strictness in the opposite direction to the PLM: "if a model element is an instance, the type has to be exactly one level above". Classifications that cross more than one level boundary go against the fundamental principles underlying PLM. Also, OMEGA allows only one type. The different possibilities for typing are not considered. This comes as no surprise, as the main purpose is code generation, a purely constructive

usecase.

The theoretical background for OMEGA and PLM is clearly the same as evidenced by their similar references. However, the elaboration of the initial ideas has progressed in almost completely opposite directions. OMEGA is intentionally not a metamodel for ontologies. PLM strives to produce ontologies which offer the same end user services as knowledge engineering technologies. OMEGA builds on top of the MOF which is not compatible with the design rationale behind PLM. So the many small differences mainly go back to the fundamentally different philosophy applied when starting from the generally equivalent base.

The research goal most completely covered by OMEGA is Research Goal 1. Goals 2 and 3 are clearly not in scope and in order to reach Research Goal 4 the realization of potency through a boolean is not sufficient. Regarding the observed weaknesses, the extension of the MOF adds more elements then there were before, so the complexity has increased rather than decreased, making Fundamental Weakness 5 the only completely covered weakness.

10.5 Nivel

Nivel(3) is a modeling language with formal semantics. The authors build on the foundations in the area of meta and multi-level modeling (9), (12), (14) or (10) shared by this work and base their language design on the principles of

- strict meta modeling,
- ontological and linguistic classification,
- unified modeling elements and
- deep classification.

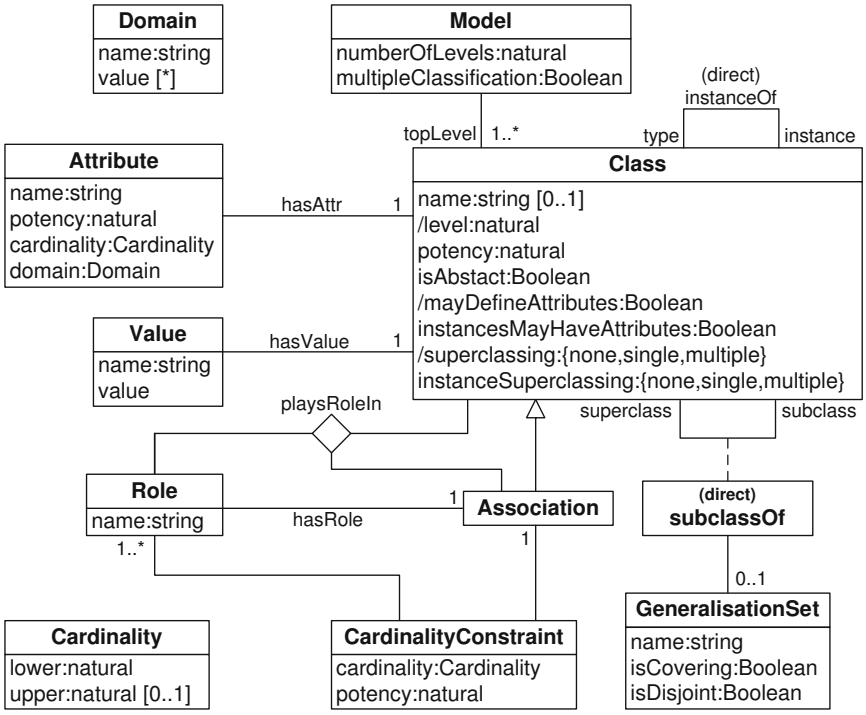


Figure 10.7: The Nivel Metamodel.(3, p.528)

The goal of Nivel is provide formal semantics for the modeling language to enable reasoning on the models. The formal semantics of Nivel is given in the

> "weight constraint rule language (WCRL), a general purpose knowledge representation language for which efficient, decidable reasoning procedures are available."(3, p. 522)

The information is input by expressing constraints. These constraints can be universally true, universally false or to be satisfied by a lower and upper cardinality. The semantics is given by defining predicates and expressing their meaning through the constraints that are sufficient. The model itself is also expressed through constraints. The total set of constraints then defines a solution space representing the valid instances of the input model

data. The WCRL interpreter (50) then finds those models and presents the output of the execution.

10.5.1 What Nivel and PLM have in common

Foundation Nivel builds on the same foundations as the PLM. Obvious proof are the properties of strict meta modeling, dual classification and potency.

Level numbering The numbering direction of the ontological levels is not just a matter of choosing to enumerate the items in a different manner. It also capture the direction by which the information is added by the user.

Association generalization Nivel treats Associations as first-class citizens of the model in the sense that they can have attributes and even enter generalization relationships. According to the Nivel metamodel (Figure 10.7) roles in Associations are played by Class which is a supertype of Association, so in Nivel, like in PLM, Associations can themselves participate in other Associations.

10.5.2 What Distinguishes Nivel and the PLM

Modeling Mode Nivel clearly takes a constructive viewpoint. The semantic definitions do not take the actual artefacts into account but take the information input as universal facts. Classification and generalization are not deduced from domain knowledge but only input in the model without verification. There are no operations to either validate statements, nor deduce statements that are not expressed but correct.

Bounded and Unbounded All the potencies in Nivel are bounded. So there is no way to express an unbound possibility for extension. Together with the rules for multiplicity and potency, many of the crucial design decisions are drawn focussed on the top level.

Limitations on ontological extension Multiplicities can only be specified at the top level. Classification is not distinguished between complete and incomplete typing.

Association participation Participating in an association in Nivel is equivalent to playing the role an association defines. Thus, the participant in an association end is not distinct. The PLM equivalent would be to have a common supertype of the clabjects playing the roles and let that be the destination of the role. However, Nivel supports roles as an independent concept in its own right. It remains unclear what the implications on the type definition of the role are. What are the properties that can be expected from the destination of the role? Are there any at all? In the example in figure 10.8, RadioPlay and Video have the same facade, but it remains unclear if only instances of the same type can play a role and whether or not they can add individual attributes to it (What if only Video added "length"?).

Multiplicity potency By assigning a potency to multiplicities, the valid number of association instances can be defined at a higher ontological level. With that mechanism, multiplicity becomes a first class citizen of the multi-level architecture as it is able to span multiple levels. In PLM, the valid number of instances can only be influenced on the directly adjacent level.

10.5.3 Conclusion

Nivel focuses on providing formal semantics to a multilevel modeling language backed by an efficient computation engine. Based on the foundations it is built on, the goal has been very well satisfied. However the goals of Nivel are even stricter than Research Goal 1. The full SE use case is not even in scope, only the formal creation of models and their translation to

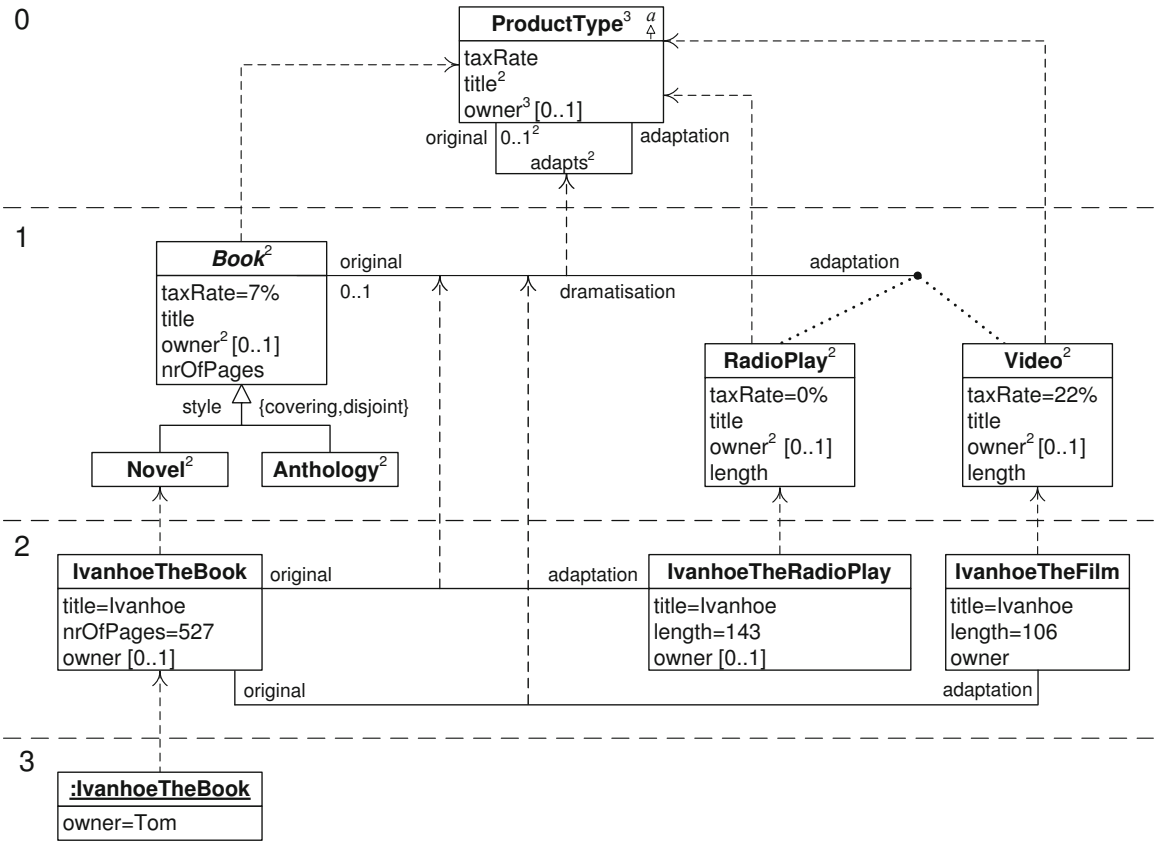


Figure 10.8: The nivel example model from (3, p. 530)

a weighted constraint language is. While this touches the use case of KE as well, it does not fully cover it. Nivel performs well in reducing accidental complexity, although not all the aspects of Research Goal 4 are solved. Especially the weaknesses 3 and 2 are not covered by Nivel.

In some of the aspects on the other hand Nivel exceeds the scope of the PLM. The playing of association roles or multiplicity potency are interesting concepts that offer expressive possibilities the current PLM does not. In the future work section in the next chapter these possibilities for the PLM are discussed.

Chapter 11

Conclusion

The goal of the research reported in this thesis was to explore and prove the validity of two hypotheses related to the use of PMLM to advance the state-of-the-art in visual modelling for software and knowledge engineering. In this chapter we summarise the outcome of this research in relation to these two hypothesis and describe the contributions that the research has made to the state-of-the-art. We also present some suggestions for further enhancements and applications of the developed technology.

11.1 Hypothesis 1

The research performed as part of this thesis has demonstrated the validity of Hypothesis 1, namely that it is feasible to enhance the original potency based multi-level modelling approach of Atkinson & Kühne in a way that:

1. supports existing SE modeling use cases by making it easier to create models with reduced accidental complexity compared to today's modelling frameworks,

2. supports existing KE modeling use cases by making it easier to create models with reduced accidental complexity compared to today's leading frameworks,

3. supports the co-use of SE and KE modelling use cases in a natural and simple way, making the uses cases traditionally available only to one of the communities available to the other as well with little if any impedance or accidental complexity,

4. supports a simple and natural approach to deep characterization, facilitating the creation of models of deep classification scenarios with less accidental complexity than today's leading modelling tools.

The key to developing a framework with the desired properties was to the combine the flexibility and visualization capabilities offered by a metamodel based modelling platform with the reasoning and analysis capabilities afforded by a formal semantics based on first-order predicate logic, using the principles laid down by the OCA(10) for PMLM. The new platform elaborated and implemented in this thesis fulfils all these requirements. In particular, it:

1. supports all the core capabilities of the leading constructive modelling technologies centred around the UML /OCL and ER(65) modelling languages,

2. supports all the core capabilities of the leading exploratory modelling technologies centred around OWL(1), including all the basic reasoning services offered by OWL based tools such as Protégé(43),

3. Allows the same model content to be leveraged in a constructive or exploratory model without any transformations or platform interchanges. Swapping between exploratory and constructive modes of modelling on the same content is therefore completely impedance free,

4. Supports an arbitrary number of ontological classification levels in a natural, level agnostic way.

11.2 Hypothesis 2

The research performed as part of this thesis has demonstrated the validity of Hypothesis 2, namely a powerful, general-purpose modelling tool on the aforementioned platform can provide additional services and uses cases beyond those supported in the two main existing modelling traditions.

The key challenge involved in exploring this hypothesis was to develop a practical modelling tool based on the aforementioned platform that makes the combined use cases of the SE and KE modelling traditions, as well as the use case cases of deep characterization, available in a and user friendly way. To meet this challenge a new multi-level modelling tool, Melanie, was developed. The PLM was implemented in EMF(24) with the help of GEF(29), GMF(30) and various other Eclipse frameworks (32, 33). This work included the definition of an enhanced concrete syntax for PMLM which seamlessly and uniformly supports all the use cases above, as well as a full implementation of all the visualization, reasoning and analysis services typically provided by tools from both modelling traditions.

From this starting point, it was possible to incorporate various new capabilities in the Melanie tool which support services and use cases that to our knowledge are not yet available anywhere else. These range from fundamental new modelling modes such as unbounded modelling (where the number of classification levels is left unspecified in a complete ontology) to very concrete model improvement and development services such as support for:

1. Instantiation from ontological types,
2. Instantiation of a complete model,
3. Validating classifications based on properties,
4. Establishing ontology properties,

5. Introspection of clabjects,

6. Refactoring of properties,

7. Removal of redundant correlations.

11.3 Contribution

In a general sense, the contribution of the thesis is to provide the foundations for a new generation of modelling tools that

(a) fulfil all the high-level goals outlined in chapter 1, namely they support all the use cases of the SE and KE modelling traditions and of deep classification within a single, unified environment and ,

(b) provide new use cases and services that are not found in any mainstream modelling tool today from either modelling tradition.

In short, it supports all existing and envisaged modelling uses cases and demonstrates the feasibility of several more.

At a more detailed level, the contributions of the research reported in thesis can best be explained from the perspective of three main constituencies – the traditional software engineering constituency, the traditional knowledge engineering constituency and the multi-level modelling research community.

11.3.1 Software Engineering

From the perspective of the SE community the work performed in this thesis makes four main significant contributions to the state of the art:

1. It supports all the main services and uses case supported by mainstream SE tools, but places them on a solid basis thanks to the developed underlying formal basis for PMLM.

2. This formal foundation allows software engineers to also leverage KE oriented use cases and services which have traditionally been unavailable to them.

3. It allows software engineers to represent deep classification scenarios in a clean, natural and uniform way, with all the traditional SE services still available to them.

4. It allows software engineers to benefit from new services that have hitherto not been available in any modelling tools.

11.3.2 Knowledge Engineering

From the perspective of the KE community the work performed in this thesis makes four main contributions to the state of the art:

1. It supports all main services and uses case supported by mainstream KE tools, but allows them to be used and leveraged with a new user-friendly engineering oriented concrete syntax akin to the UML. In short, it let's user visualize ontologies though a user friendly yet engineering-oriented modeling notation.

2. It allows knowledge engineers to also leverage SE oriented use cases and service which have traditionally been unavailable to them without prohibitive artificial complexity.

3. It allows knowledge engineers to represent deep classification scenarios in a clean, natural and uniform way, with all the traditional KE services.

4. It allows knowledge engineers to benefit from new services that have hitherto not been available in any modelling tools.

11.3.3 Multi-Level Modeling

From the perspective of the PMLM community the work performed in this thesis makes four main significant contributions to the state of the art:

1. It extends the original PMLM approach of Atkinson & Kühne to support all the main uses cases of the SE and KE modelling communities as well as PMLM and new hitherto unknown modelling use cases.

   More specially, the key enhancements to the original PMLM approach of Atkinson & Kühne that enable this are:

   Star potencies to enable the new dimension of unbounded modeling,

   Isonyms and Hyponyms to refine the notion of classification and its role in exploratory and constructive modeling,

   Formal Correlations to validate asserted information and discover correlations based on user input,

   Value potency to introduce multi-level control not only for clabjects and features, but also for values,

   Multi-level Correlations To formalize the semantics of correlations with their meaning for the classified domain and can be validated and enforced by the modeling tool,

   Level spanning reasoning Now that the multi-level traits and concepts are formalized, there are ontology properties (consistency, completeness) that span multiple levels and thus bring logical reasoning to the multi-level world.

2. It has provided one of the first comprehensive formal foundations for PMLM along with a comprehensive, complementary set of reasoning services that encompass those of traditional modelling environments,

3. It has developed the first comprehensive, visual concrete syntax to support PMLM as well as the new use cases and services mentioned above,

4. It has developed one of the first practical tools for PMLM, featuring all the aforementioned innovations, that is compatible with the most widespread open source modeling frameworks available today (i.e. Eclipse, EMF).

11.3.4 Prototype Implementation – Melanie

The ideas presented in this thesis have been implemented in a modeling tool that strives to fulfil the research goals formulated in chapter 1. The tool is called *Melanie* for "Multi-Level Modeling and Ontology Engineering Environment"(33), (32).

The tool is implemented on top of the Eclipse framework and makes heavy use of the provided plugins to benefit as much as possible from previous work. The metamodel is realized in EMF(28). The metamodel operations are implemented in the metamodel as OCL(54) operations. Therefore they are also accessible in the interactive OCL console for live queries to the ontology (see figure 11.1, 11.2). The editor is implemented in GMF(30).

The editor features a palette from which concrete metamodel elements can be instantiated. Connections can be drawn between the entities. For types the palette is extended by domain specific elements. The appearance of these elements can be configured. Figure 11.1 shows the editor with an almost empty ontology. On the left hand side is the project explorer were the ontologies can be organized into projects. The outline below makes it easier to navigate through large ontologies. At the bottom is the properties view displaying all the traits, but also ontological and visualization information. In the bottom-right corner there is an open interactive OCL console where the user can query the model and get immediate feedback.

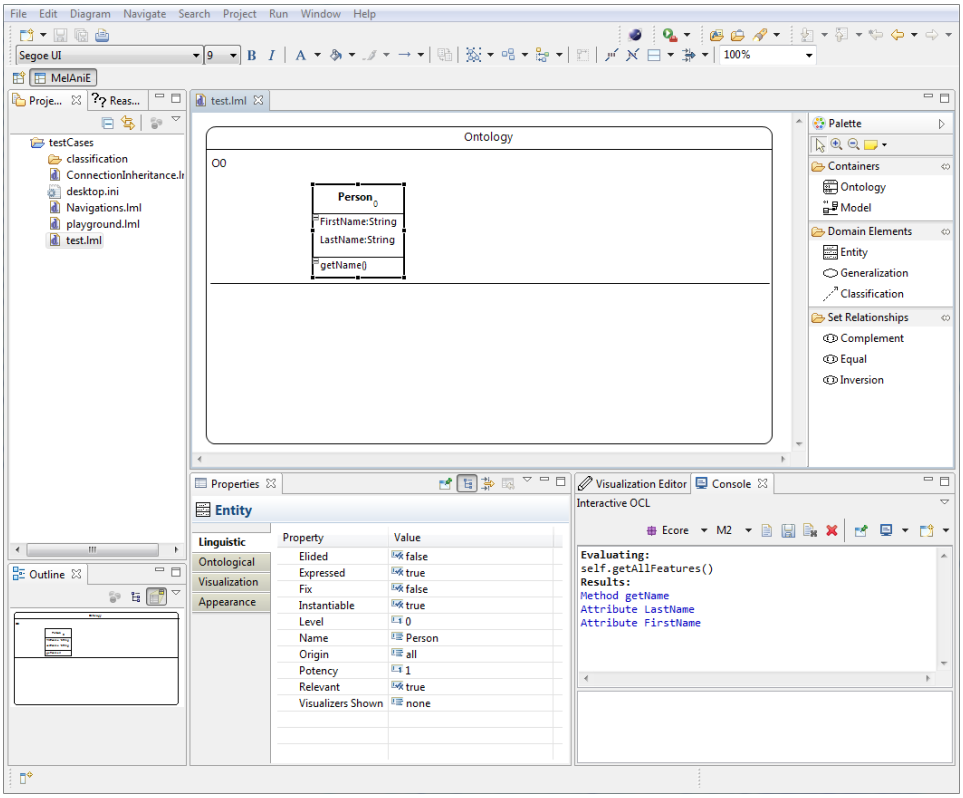


Figure 11.1: Screenshot of an almost empty ontolgoy in Melanie

The second screenshot (figure 11.2) shows the result of a reasoning query in Melanie. In O1 there is an anonymous entity without a type and the user has requested the tool to check whether it is an instance of *Person* via the context menu of the elements. The result can be seen on the left hand side in the reasoning view. The result is not just a boolean answer ("yes" or "no") but a detailed trail of the checks performed in order to obtain the result. In this example the entity is a person because it is an isonym (i.e. it property conforms, potency conforms and does not add any new properties). The reasoning view provides valuable feedback about the origin of the overall result, especially when the overall check fails and the user wants to investigate the cause of the failure.

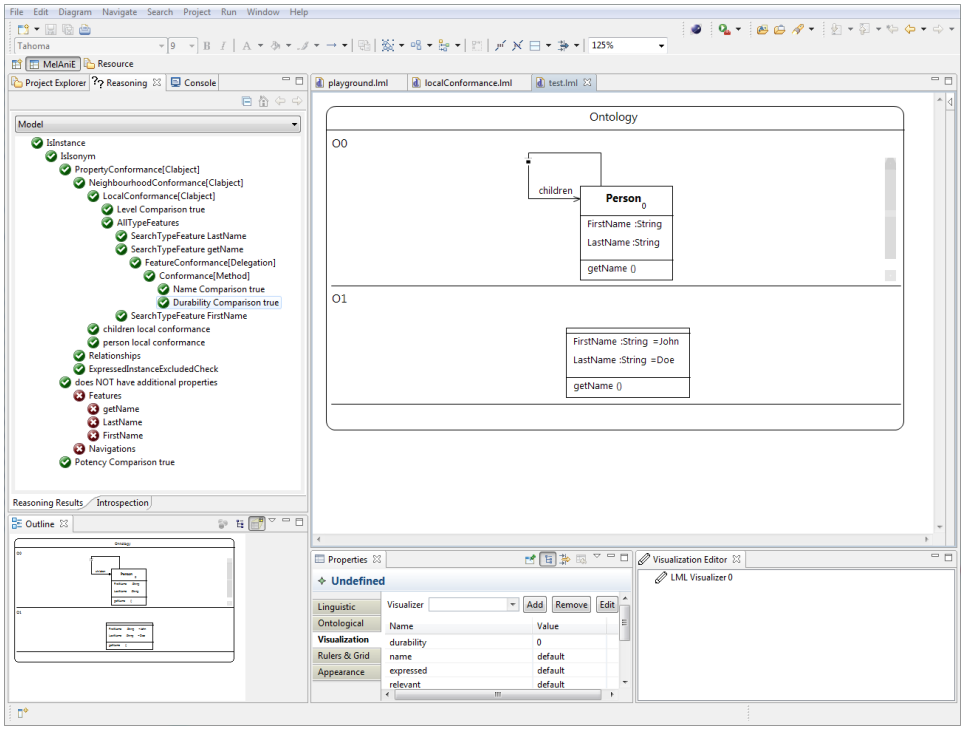


Figure 11.2: Screenshot of a reasoning result in Melanie

The third screenshot (figure 11.3) shows a larger ontology. The left hand side and the bottom panel have been minimized to provide an editor centric working environment. In the lower ontological level, a domain specific rendering of one of the entities has been selected. One of the main features of Melanie is the symbiotic usage of both domain specific and general purpose rendering, as shown in O2. The elements in O2 were created by ontological instantiation of previously defined types. Melanie offers those types in the DSL Palette (shown on the right hand side in figure 11.3) as soon as the lower level is selected. The rendering and domain specific capabilities are the focus of another branch of research ongoing at the group of software engineering (17).

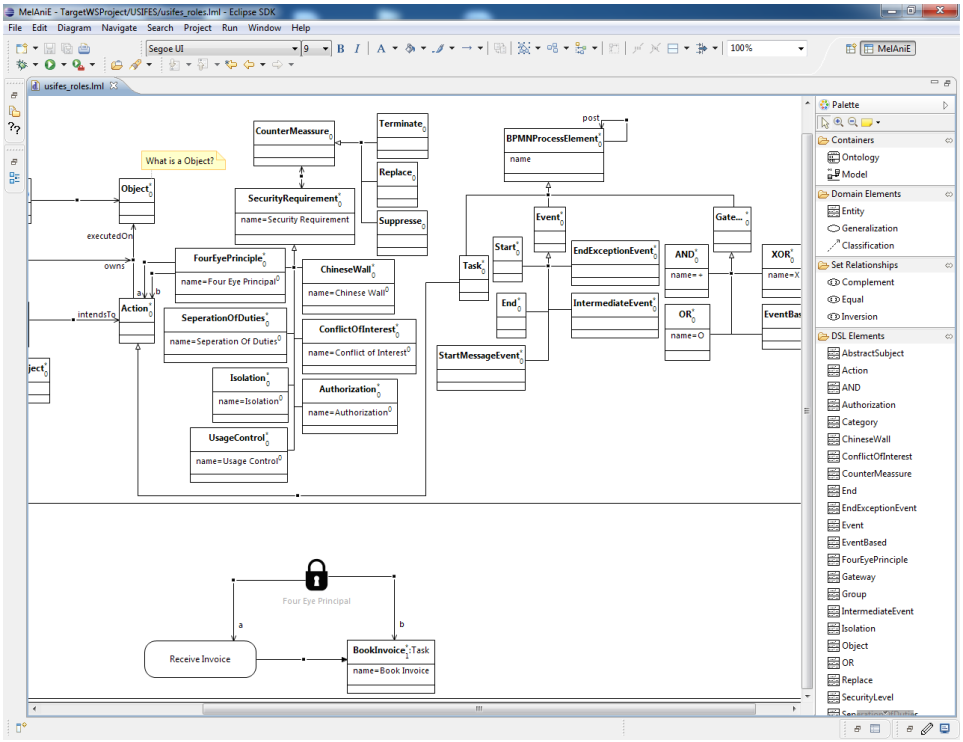


Figure 11.3: Screenshot of a larger ontology in Melanie containing domain specific rendering.

Despite the effort put into the development of Melanie, it is still a prototype implementation. As such it is not optimized for fast computation but to show the power and possibilities of the technology. While the current version is certainly usable, it cannot operate on large ontologies of the kind found on the semantic web. The bottleneck is the visual rendering of the clabjects. The EMF technology is able to operate on large models, as are all the introduced services.

With the recursive nature of many of the introduced definitions (mainly property conformance), the execution time of a reasoning operation such as subsumption or ontology validation rises rapidly with the number of involved clabjects. Analyses of the implementation shows that the worst case runtime

for exploratory ontology validation is $O(n^8)$. The actual runtime is highly dependent on the shape of the ontology. The tighter the integration of the clabjects through connections, the more checks need to be executed.

11.4 Future Work

Beyond the benefits just described, the presented work also opens up a lot of potential for further research. The proposed research can be divided into enhancing the present solution to resolve shortcomings and enhancements to add new features.

11.4.1 Evolutionary Enhancements

The enhancements presented in this subsection are called evolutionary because they build on the present solution and present only an incremental improvement.

Composition and Containment in general The current version of PLM does not have a distinct metamodel element for containment in general or composition in particular. The two possible alternatives are: a subtype of connection or a correlation.

Packages The current version of the PLM metamodel already combines elements of three different dimensions: Rendering information, artefacts and correlations. If these were separated by packages, they could be imported and exported, making metamodel development and refinement much more structured.

Mutability for Methods Mutability does a good job of fixing the value of attributes from being overridden on classified levels. The equivalent to attribute values for methods is their body. A similar concept for the body does not exist but would be very powerful.

Separation of correlations and artefacts Generalizations carry ontological information because the properties are not passed on to the subtypes. That makes generalization a part of the domain definitions which may not be desirable. If generalization would be processed in a way that the properties are created in the subtypes upon the creation of the generalization, all artefacts would be present in the namespace of any clabject.

Connection Transitivity The transitivity trait of connections is not used in reasoning at the moment. If a connection δ is transitive and two isonyms δ_1, δ_2 have the participants γ_1, γ_2 and γ_3 such that $\gamma_1\vec{\delta_1}\gamma_2$ and $\gamma_2\vec{\delta_2}\gamma_3$ then there must exist a third isonym $\gamma_1\vec{\delta_3}\gamma_3$. The current version of the engine does support the detection or the enforcement of transitivity.

Overriding of Navigations and Connections Each clabject can participate in connections. Connection participation is a property that is passed on to the subtypes. There is no formalization for identifying overridden navigations. The following approach in operation 15 could provide a method to identify one kind of redefined roles, but has not been implemented or tested.

Operation 15 γ.`redefinedNavigations`()

```
result ← ∅
for ψ ∈ γ.inheritedNavigations() do
  δ ← ξ.connection
  γ_s ← ξ.destination
  for ψ' ∈ γ.eigenNavigations() : ψ.roleName = ψ'.roleName do
    if ψ'.connection.subsume(δ) ∧ ψ'.destination.subsume(γ_s) then
      result ← result ∪ {ψ}
return result
```

Playing of association roles The current semantics for roles is that if a clabject participates in a connection, its instances have to participate in an instance of that connection as well. If a role could define several clabjects that are valid types for connection participants, the semantic would change to: An instance of one of the types that play that role has to participate in any instance of the connection.

Multiplicity potency Among the traits that are multi-level unaware is the name of clabjects and multiplicity. If multiplicity had a potency like features, there would be two consequences: Σ_i connections could influence the shape of Σ_{i+2} directly and Σ_{i+1} could be actively flagged as an intermediate level not subject to multiplicity constraints.

Textual syntax The EMF modeling framework(28) has an internal representation supporting a textual syntax, but the models are far from human readable, meaning the model loses the benefit of the graphical simplicity. Ideally, a model would be human readable in both textual and graphical syntax.

11.4.2 Ontology Properties

As well as consistency, completeness and validity ontologies can be assigned other properties that show their maturity or logical soundness.

Contained Ontology Consistency and completeness both judge top-down that every declared type actually is a type and that the offspring conforms to the claims made. A self contained ontology is an ontology that contains all the information necessary to produce itself. So every model except the root model can be created from its classifying model. Formally, in a self contained ontology every clabject has a complete type.

$$\chi.\texttt{isSelfContained}() := \chi.\texttt{isComplete}() \wedge \forall \gamma : \gamma.\mathsf{level} \neq i_m in :$$

$\exists \gamma_t : \gamma_t.\texttt{isCompleteType}(\gamma)$

Minimal Ontology Ideally, the upper ontological levels server the purpose of specifying the lower levels. Of course there may be clabjects not on the leaf model that have no relationships to lower levels, but then these clabjects are either not necessary for the specification of the most concrete domain or the ontology is not complete. In a minimal ontology, every element is necessary for the classification of the lower ontological levels. Formally it means that every clabject (except for the leaf model) has a potency greater than zero.

$\chi.\texttt{isMinimal}() := \chi.\texttt{isComplete}() \wedge \forall \gamma : \gamma.\mathsf{level} \neq i_m ax :$
$\gamma.\mathsf{potency} > 0$

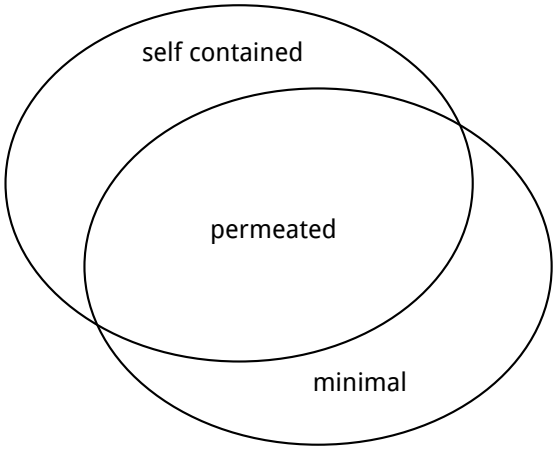
Permeated Ontology Even with a minimal ontology, it is still possible that some concepts are not introduced in the root model but further down the classification hierarchy. To describe an ontology where every concept is present on all levels, a permeated ontology is introduced. Every clabject permeates the whole ontology in the sense that its classification path goes from the root to the leaf model. Formally, every potency at the top level is equivalent to the number of classified models.

$\chi.\texttt{isPermeated}() := \chi.\texttt{isComplete}() \wedge \forall \gamma :$
$\gamma.\mathsf{potency} = \chi.\texttt{levels}() - \gamma.\mathsf{level}$

In a permeated ontology, the models can still be of different size. It is even possible for a classified model to be smaller in size than the classifying model, but the most likely case is that the size of the models increases with the level.

Status These properties are not yet implemented in Melanie but are targeted for future versions. Although the properties form a sequence of strict-

Figure 11.4: Landscape of ontology properties



ness in their requirements, they are not complete specializations. Figure 11.4 gives an overview of the property implications. Every permeated ontology is self contained and minimal. There can be not permeated minimal ontologies as well as not permeated self contained ones. An ontology can be minimal and not self contained and vice versa, but once it is minimal and self contained, it is permeated.

11.4.3 Open World and Closed World

If the information in the model is not meant to be complete, the LML (16) is already able to reflect this fact in the concrete syntax by applying elision. The elided part in the open world context stands for the unknown part of the information.

With the possibility of [0..1] multiplicities for PLM traits, the infrastructure supports undefined traits. This lack of definition can be interpreted as "unspecified" rather than "missing" or "universally true" instead of "tied to one value". These measures aim at supporting open world reasoning at some point in the future. Supporting open world reasoning is not as easy as switching from binary to ternary logic. The whole semantics of the model

element (may) differ in the open world. The challenge comprises

a) model repository representation(e.g. using non-mandatory traits),

b) visual rendering (e.g. using elision) and

c) semantic operation definition both in terms of open and closed world semantics.

While there is still a lot of work to be done to lay the ground for the last part, the PLM tries to be compatible with an implementation of open and closed world distinction in the future.

The presentation of these opportunities for future work brings the thesis to its conclusion. Hopefully the technology described in the preceding chapters will help lay the foundation for a new generation of tools that are able to bridge the barrier that currently divides the software engineering and knowledge engineering communities and will lower the artificial complexity that modelers currently have to contend with when developing models. At the very least, the ideas and discussions will hopefully stimulate further PhD students to push forward the state-of-the-art in modeling, so that one day modeling will become the only important activity in software and knowledge engineering.

Bibliography

[1] Grigoris Antoniou and Frank Van Harmelen. OWL Web Ontology Language. *Ubiquity*, 2007(September):1–1, 2004. ISSN 15302180. doi: 10.1145/1295289.1295290. URL `http://portal.acm.org/citation.cfm?doid=1295289.1295290`. 30, 242

[2] T Aschauer, G Dauenhauer, and W Pree. Multi-level Modeling for Industrial Automation Systems. In *Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on*, pages 490–496, 2009. doi: 10.1109/SEAA.2009.46. 34

[3] Timo Asikainen and Tomi Männistö. Nivel: a metamodelling language with a formal semantics. *Software & Systems Modeling*, 8 (4):521–549, November 2008. ISSN 1619-1366. doi: 10.1007/s10270-008-0103-2. URL `http://www.springerlink.com/index/10.1007/s10270-008-0103-2`. 34, 45, 236, 237, 240

[4] Uwe Assmann, Steffen Zschaler, and Gerd Wagner. Ontologies, Meta-models, and the Model-Driven Paradigm. *Ontologies for Software Engineering and Software Technology*, pages 249–273, 2006. doi: 10.1007/3-540-34518-3_9. URL `http://scholar.google.com/scholar?num=100&#38;hl=en&#38;lr=&#38;q=%22Ontologies%2C+Meta-models%2C+and+the+Model-Driven+Paradigm%22&#38;btnG=Search`. 33

[5] C Atkinson. Meta-modelling for distributed object environments. In *Enterprise Distributed Object Computing Workshop [1997]. EDOC'97. Proceedings. First International*, pages 90–101. IEEE, 1997. 34, 42, 45

[6] Colin Atkinson. Supporting and applying the UML conceptual framework. *The Unified Modeling Language.UML98: Beyond the Notation*, page 514, 1999. 43, 45

[7] Colin Atkinson and Kilian Kiko. A Detailed Comparison of Uml and Owl. 2008. 30

[8] Colin Atkinson and Thomas Kühne. Meta-level independent modelling. In *International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming*, pages 12–16. Citeseer, 2000. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.26.3964&rep=rep1&type=pdf`. 45

[9] Colin Atkinson and Thomas Kühne. The essence of multilevel metamodeling. *UMLThe Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 19–33, 2001. URL `http://www.springerlink.com/index/ccbgph1thqmx9myn.pdf`. 43, 44, 45, 118, 220, 236

[10] Colin Atkinson and Thomas Kühne. Rearchitecting the UML infrastructure. *ACM Transactions on Modeling and Computer Simulation*, 12(4):290–321, October 2002. ISSN 10493301. doi: 10.1145/643120.643123. URL `http://portal.acm.org/citation.cfm?id=643120.643123`. 34, 41, 45, 236, 242

[11] Colin Atkinson and Thomas Kühne. Profiles in a strict metamodeling framework. *Science of Computer Programming*, 44(1):5–22, July 2002. ISSN 01676423. doi: 10.1016/S0167-6423(02)00029-1. URL `http://linkinghub.elsevier.com/retrieve/pii/S0167642302000291`. 45

[12] Colin Atkinson and Thomas Kühne. Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5):36–41, September 2003. ISSN 0740-7459. doi: 10.1109/MS.2003.1231149. URL http://www.computer.org/portal/web/csdl/doi/10.1109/MS.2003.1231149. 34, 45, 220, 236

[13] Colin Atkinson and Thomas Kühne. Reducing accidental complexity in domain models. *Software and Systems Modeling*, 7(3):345–359, 2008. ISSN 1619-1366. URL http://dx.doi.org/10.1007/s10270-007-0061-0. 190

[14] Colin Atkinson, Thomas Kühne, and Brian Henderson-Sellers. Stereotypical Encounters of the Third Kind. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *UML 2002 The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 329–334. Springer Berlin / Heidelberg, 2002. URL http://dx.doi.org/10.1007/3-540-45800-X_9. 45, 236

[15] Colin Atkinson, Matthias Gutheil, and Bastian Kennel. A Flexible Infrastructure for Multilevel Language Engineering. *IEEE Transactions on Software Engineering*, 35(6):742–755, November 2009. ISSN 0098-5589. doi: 10.1109/TSE.2009.31. URL http://www.computer.org/portal/web/csdl/doi/10.1109/TSE.2009.31. 43, 220

[16] Colin Atkinson, Bastian Kennel, and Björn Goß. The Level-Agnostic Modeling Language. In *Software Language Engineering*, pages 266—-275. Springer, 2011. URL http://www.springerlink.com/index/K335GV307412743G.pdf. 36, 109, 112, 255

[17] Colin Atkinson, Ralph Gerbig, and Bastian Kennel. Symbiotic General-Purpose and Domain-Specific Languages. In *Proceedings of the 34rd International Conference on Software Engineering, ICSE 2012*. ACM, 2012. 249

[18] Sean Bechhofer, Frank Van Harmelen, Jim Hendler, Ian Horrocks, Deborah L McGuinness, Peter F Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontology Language Reference, 2004. URL `http://www.w3.org/TR/owl-ref/`. 30

[19] D Berardi, D Calvanese, and G Degiacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168(1-2):70–118, October 2005. ISSN 00043702. doi: 10.1016/j.artint.2005.05.003. URL `http://linkinghub.elsevier.com/retrieve/pii/S0004370205000792`. 33

[20] Artur Boronat and Jose Meseguer. An Algebraic Semantics for MOF. In José Luiz Fiadeiro and Paola Inverardi, editors, *Formal Aspects of Computing*, volume 22 of *Lecture Notes in Computer Science*, pages 269–296. Springer., 2009. URL `http://www.springerlink.com/content/w43259838k7h/?sortorder=asc&p_o=20`. 218

[21] T Bray, J Paoli, C M Sperberg-McQueen, E Maler, F Yergeau, and Others. Extensible markup language (XML) 1.0, 2000. 31

[22] F P Brooks Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987. 32

[23] F Budinsky. *Eclipse modeling framework: a developer's guide*. Addison-Wesley Professional, 2004. 34

[24] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse modeling framework: a developer's guide*. Addison-Wesley Professional, 2004. 229, 243

[25] Christoff Bürger, Sven Karol, Christian Wende, and Uwe Assmann. Reference attribute grammars for metamodel semantics. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 22–41. Springer Berlin / Heidelberg, 2011. 218

[26] Peter Pin-Shan Chen. The entity-relationship modeltoward a unified view of data. *ACM Transactions on Database Systems TODS*, 1(1): 9–36, 1976. ISSN 03625915. doi: 10.1145/320434.320440. URL `http://portal.acm.org/citation.cfm?id=320440&dl=\unhbox\voidb@x\bgroup\accent127U\penalty\@M\hskip\z@skip\egroup`. 30

[27] Peter Chen Computer and Peter P. Chen. Entity-Relationship Modeling: Historical Events, Future Trends, and Lessons Learned, 2002. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.15.4630`. 30

[28] Eclipse Foundation. Eclipse Modeling Framework Project, . URL `http://www.eclipse.org/modeling/emf/`. 247, 253

[29] Eclipse Foundation. Graphical Editing Framework, . URL `http://www.eclipse.org/gef/`. 229, 243

[30] Eclipse Foundation. Graphical Modeling Framework (GMF), . URL `http://www.eclipse.org/modeling/gmp/`. 229, 243, 247

[31] Eclipse Foundation. Xtext, . URL `http://www.eclipse.org/Xtext/`. 229

[32] Ralph Gerbig. The Level-agnostic Modeling Language: Language Specification and Tool Implementation, 2011. 109, 243, 247

[33] Ralph Gerbig and Bastian Kennel. Multi level modeling and ontology engineering environment, 2011. URL `http://code.google.com/a/eclipselabs.org/p/melanie/`. 227, 243, 247

[34] R. Gitzel, I. Ott, and M. Schader. Ontological Extension to the MOF Metamodel as a Basis for Code Generation. *The Computer Journal*, 50(1):93–115, October 2006. ISSN 0010-4620. doi: 10.1093/

comjnl/bxl052. URL http://comjnl.oxfordjournals.org/cgi/doi/10.1093/comjnl/bxl052. 34, 233, 235

[35] Martin Gogolla, Jean-Marie Favre, and Fabian Büttner. On squeezing M0, M1, M2, and M3 into a single object diagram, 2005. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.103.806. 33

[36] Cesar Gonzalez-Perez and Brian Henderson-Sellers. A powertype-based metamodelling framework. *Software and Systems Modeling*, 5 (1):72–90, 2006. ISSN 1619-1366. URL http://dx.doi.org/10.1007/s10270-005-0099-9. 33, 45

[37] Matthias Gutheil, Bastian Kennel, and Colin Atkinson. A Systematic Approach to Connectors in a Multi-level Modeling Environment. *Model Driven Engineering Languages and Systems*, pages 843–857, 2008. URL http://www.springerlink.com/index/B64168K41LX862WX.pdf. 43

[38] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3):301–317, 2000. 218

[39] B. Henderson-Sellers. Bridging metamodels and ontologies in software engineering. *Journal of Systems and Software*, 84(2):301–313, February 2011. ISSN 01641212. doi: 10.1016/j.jss.2010.10.025. URL http://portal.acm.org/citation.cfm?id=1922690.1922987. 33

[40] M Horridge, H Knublauch, A Rector, R Stevens, and C Wroe. A Practical Guide To Building OWL Ontologies Using The Protege-OWL Plugin and CO-ODE Tools Edition 1.4. *The University Of Manchester*, 2004. 31, 189, 192, 201

[41] Nophadol Jekjantuk, G. Gröner, J.Z. Pan, and Y. Zhao. Modelling and Validating Multilevel Models with OWL FA. In *Proceedings of the 6th*

International Workshop on Semantic Web Enabled Software Engineering, 2010. URL http://www.uni-koblenz.de/~groener/documents/SWESE2010_OWLFA.pdf. 33, 46

[42] Juan A. Recio-García. OntoBridge, 2008. URL http://gaia.fdi.ucm.es/research/ontobridge. 50

[43] H Knublauch, R Fergerson, N Noy, and M Musen. The Protege OWL plugin: An open development environment for semantic web applications. *The Semantic Web–ISWC 2004*, pages 229–243, 2004. 31, 189, 242

[44] D Kolovos, R Paige, and F Polack. The epsilon object language (eol). In *Model Driven Architecture–Foundations and Applications*, pages 128–142. Springer, 2006. 220

[45] Thomas Kühne. Matters of (Meta-) Modeling. *Software Systems Modeling*, 5(4):369–385, 2006. ISSN 16191366. doi: 10.1007/s10270-006-0017-9. URL http://www.springerlink.com/index/10.1007/s10270-006-0017-9. 111

[46] Thomas Kühne and F. Steimann. Tiefe charakterisierung. In *Modellierung 2004*, pages 121–133. Ges. für Informatik, 2004. ISBN 3885793741. URL http://deposit.fernuni-hagen.de/2248/. 37

[47] Juan De Lara. Deep meta-modelling with METADEPTH. *Objects, Models, Components, Patterns*, pages 1–18, 2010. URL http://www.springerlink.com/index/1034254L64508344.pdf. 45, 220, 222

[48] Barbara Liskov. Data abstraction and hierarchy. In *Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, OOPSLA '87, pages 17–34, New York, NY, USA, 1987. ACM. ISBN 0-89791-266-7. doi: http://doi.acm.org/

10.1145/62138.62141. URL http://doi.acm.org/10.1145/62138.62141. 136

[49] D L McGuinness, F Van Harmelen, and Others. OWL web ontology language overview. *W3C recommendation*, 10:2003–2004, 2004. 30

[50] Ilkka Niemelä and Patrik Simons. Smodels an implementation of the stable model and well-founded semantics for normal logic programs. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Logic Programming And Nonmonotonic Reasoning*, volume 1265 of *Lecture Notes in Computer Science*, pages 420–429. Springer Berlin / Heidelberg, 1997. URL http://dx.doi.org/10.1007/3-540-63255-7_32. 238

[51] Object Management Group. Unified Modeling Language (OMG UML), Infrastructure Version 2.3, 2010. URL http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/. 30, 38

[52] J J Odell. *Advanced object-oriented analysis and design using UML*, volume 12. Cambridge Univ Pr, 1998. 39

[53] OMG. Unified Modeling Language v. 1.1. *OMG document ad/97-08-11*, 1997. 109

[54] OMG. Object Constraint Language, v2.2. *Management*, 03(February), 2010. 63, 247

[55] OMG. Meta Object Facility (MOF) 2.4.1 Core Specification Version, 2011. URL http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:OMG+Meta+Object+Facility+(+MOF+)+Core+Specification#4. 217, 218, 228, 233

[56] OMG. OMG Unified Modeling Language TM (OMG UML), Superstructure, 2011. 30

[57] OMG. OMG MOF 2 XMI Mapping Specification. (August), 2011. 39

[58] F S Parreiras, S Staab, and A Winter. TwoUse: Integrating UML models and OWL ontologies. *Universit{\"a}t Koblenz-Landau, Fachbereich Informatik*, 2007. 33

[59] F.S. Parreiras, S. Staab, and A. Winter. On marrying ontological and metamodeling technical spaces. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 439–448. ACM, 2007. 50

[60] A Pirotte, E Zimányi, D Massart, and T Yakusheva. Materialization: a powerful and ubiquitous abstraction pattern. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 630–641. Morgan Kaufmann Publishers Inc., 1994. 45

[61] Iman Poernomo. The meta-object facility typed. *Proceedings of the 2006 ACM symposium on Applied computing SAC 06*, 06pages:1845–1849, 2006. doi: 10.1145/1141277.1141710. URL `http://portal.acm.org/citation.cfm?id=1141710`. 218

[62] Jan Reimann, Mirko Seifert, and Uwe Assmann. Role-Based Generic Model Refactoring. In Dorina Petriu, Nicolas Rouquette, and Ø ystein Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6395 of *Lecture Notes in Computer Science*, pages 78–92. Springer Berlin / Heidelberg, 2010. URL `http://dx.doi.org/10.1007/978-3-642-16129-2_7`. 167

[63] F Rieckhof, M Seifert, and Uwe Assmann. Ontology-based Model Synchronization. *Third Workshop on Transforming and Weaving OWL Ontologies in MDE/MDA (TWOMDE 2010)*, page 15, 210. 50

[64] Lijun Shan and Hong Zhu. Semantics of Metamodels in UML. *2009 Third IEEE International Symposium on Theoretical As-*

pects of Software Engineering, 0:55–62, 2009. URL http://doi.ieeecomputersociety.org/10.1109/TASE.2009.62. 218

[65] Bernhard Thalheim. Entity-Relationship Modeling: Foundations of Database Technology. January 2000. URL http://0-portal.acm.org.millennium.lib.cyut.edu.tw/citation.cfm?id=555645. 242

[66] D Varró and A Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Software and Systems Modeling*, 2(3):187–210, 2003. 34

[67] B Volz. A meta model for representing arbitrary meta model hierarchies. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2371–2372. ACM, 2010. 227

[68] B Volz. *Werkzeugunterstützung für Methodenneutrale Metamodellierung*. PhD thesis, University of Bayreuth, Bayreuth, Germany, July 2011. 227

[69] B Volz and S Jablonski. OMME–A Flexible Modeling Environment. In *Proceedings of SPLASH Workshop on Flexible Modeling Tools (FlexiTools)*, 2010. 34, 45, 227, 230

[70] B Volz and S Jablonski. Towards an open meta modeling environment. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*, page 17. ACM, 2010. 227, 228, 229

[71] B Volz, M Zeising, and S Jablonski. The Open Meta Modeling Environment. *ICSE 2011 Workshop on Flexible Modeling Tools (FlexiTools 2011)*, 2011. 227

[72] W3C Recommendation. RDF Vocabulary Description Language 1.0: RDF Schema, 2004. URL http://www.w3.org/TR/2004/REC-rdf-schema-20040210/. 30

[73] J Warmer and A Kleppe. *The object constraint language: getting your models ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., 2003. 63, 204

[74] J B Warmer and A G Kleppe. The Object Constraint Language: Precise Modeling With Uml (Addison-Wesley Object Technology Series). 1998. 63

[75] Claas Wilke, Sebastian Götz, Jan Reimann, and Uwe Assmann. Vision paper: Towards model-based energy testing. In *Proceedings of 14th International Conference on Model Driven Engineering Languages and Systems (MODELS 2011)*, 2011. 217