University of Mannheim

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

Diploma Thesis at the University of Mannheim in Wirtschaftsinformatik

Supervisor: Bastian Kennel

The Level-agnostic Modeling Language: Language Specification and Tool Implementation

Ralph Gerbig
<rgerbig@rummss.uni-mannheim.de>

Mannheim, May 2011

Abstract

Since the release of the Entity-Relationship modelling language in 1976 and the UML in the early 1990's no fundamental developments in the concrete syntax of general purpose modelling languages have been made. With today's trends in model-driven technologies and the rising need for domain specific languages the weaknesses of the traditional languages become more and more obvious. Among these weaknesses are missing support for modelling multiple ontological levels or the lack of built-in Domain Specific Language development capabilities. The Level-agnostic Modeling Language (LML) was developed to address these two needs. During its development care was taken to retain the strengths of traditional languages.

This thesis is based on a collection of papers about multilevel modelling. The collection starts with a paper that identifies the need for multilevel modelling through a practical example of a language used to describe computer hardware product hierarchies. A later paper examines the problems of current technologies from a more theoretical point of view and suggestions to solve the identified issues are made. The latest work in this collection defines the LML based on previously made observations. The work on the LML has now reached a maturity level which makes it worthwhile to write an LML specification 1.0 and implement a tool to give other researchers the opportunity to use this new technology.

The thesis provides the specification 1.0 of the LML. Additionally, a graphical editor based on one of today's leading model driven development platforms, Eclipse, is developed.

Contents

Glossary vi

1. Introduction 1

2. Foundations 3

2.1. Model-Driven Software Development . . . 3
2.1.1. Model-Driven Software Development Definitions . . . 3
2.1.2. Meta-Modelling . . . 4
2.1.3. Domain Specific Languages . . . 6
2.1.4. Model Transformations . . . 7
2.1.5. The Level-agnostic Modeling Language . . . 8
2.2. DSL Modelling with Eclipse . . . 10
2.2.1. Eclipse Rich Client Platform . . . 10
2.2.2. Eclipse Modelling Framework . . . 11
2.2.3. Graphical Editing Framework . . . 12
2.2.4. Graphical Modelling Framework . . . 12
2.2.5. Model Constraint Languages in GMF and EMF . . . 14
2.2.6. Model Transformations in GMF and EMF . . . 15

3. The Level-agnostic Modeling Language Specification 16

3.1. Abstract Syntax . . . 16
3.1.1. Modelling Modes . . . 17
3.1.2. Element . . . 18
3.1.3. Ontology, Model and OwnedElement . . . 19
3.1.4. LogicalElement . . . 19
3.1.5. DomainElement . . . 22
3.1.6. Clabject . . . 22
3.1.7. Feature . . . 25
3.1.8. VisualizationContainer and TopLevelVisualizationContainer . . . 26
3.1.9. Visualizer . . . 27
3.2. Concrete Syntax . . . 28
3.2.1. Default Value Handling by the Concrete Syntax . . . 28
3.2.2. Trait Value Specification . . . 28
3.2.3. Visualizer . . . 28
3.2.4. Ontology and Model . . . 29
3.2.5. Entity . . . 29
3.2.6. Connection . . . 30

3.2.7. Proximity Indication for Clabjects . . . 31
3.2.8. Dottability of Relationships . . . 32
3.2.9. Elision . . . 32
3.2.10. Feature . . . 33
3.2.11. Instantiation . . . 33
3.2.12. Set Relationship . . . 34
3.2.13. Generalization . . . 35

4. The Level-agnostic Modeling Language Editor Implementation 37
4.1. The Level-agnostic Modeling Language Editor . . . 37
4.2. Abstract Syntax Implementation . . . 38
4.3. Diagram Editor Implementation . . . 40

5. Level-agnostic Modeling Language Examples 45
5.1. The Pizza Ontology Example . . . 45
5.2. The Entity-Relationship Diagram Example . . . 46
5.3. The Java Enterprise Edition Profile Example . . . 46
5.4. The Royal & Loyal OCL Example . . . 48

6. Future Work 49
6.1. Abstract and Concrete Syntax . . . 49
6.2. The Level-agnostic Modeling Language Editor . . . 49
6.3. Domain Specific Language Engineering Support . . . 50
6.4. Deep Transformation, Constraint and Query Language . . . 55

7. Related Work 59

8. Conclusion 60

A. LML Editor User Manual 65
A.1. Installation . . . 65
A.2. Walkthrough: Creating a Diagram File . . . 66
A.3. Walkthrough: The First Ontology . . . 68
A.4. Walkthrough: Using Visualizers . . . 72

Ehrenwörtliche Erklärung 75

Abtretungserklärung 76

Glossary

API	Application Programming Interface
ATL	ATLAS Transformation Language
DSL	Domain Specific Language
DSML	Domain Specific Modelling Language
EMF	Eclipse Modelling Framework
EMOF	Essential Meta-Object Facility
EMP	Eclipse Modelling Project
ER	Entity-Relationship
GEF	Graphical Editing Framework
GMF	Graphical Modelling Framework
GMP	Graphical Modelling Project
GPL	General Purpose Language
HOT	Higher-Order-Transformation
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
J2EE	Java Enterprise Edition
LML	Level-agnostic Modeling Language
M2M	Model-to-Model
M2T	Model-to-Text
MBSD	Model-Based Software Development
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MDSD	Model-Driven Software Development
MOF	Meta Object Facility
MVC	Model View Controller
OCA	Orthogonal Classification Architecture
OCL	Object Constraint Language
OMG	Object Management Group
OOPL	Object Oriented Programing Language
OWL	Web Ontology Language
PLM	Pan-Level Meta-Model
QVTo	Query View Transformation Language operational
RCP	Rich Client Platform
SVG	Scalable Vector Graphics
TVS	Trait Value Specification
UI	User Interface

UML	Unified Modeling Language
WPF	Windows Presentation Foundation
WYSIWYG	What You See Is What You Get
XAML	Extensible Application Markup
XMI	XML Metadata Interchange
XML	Extensible Markup Language

List of Figures

1. A Meta-Model Example 4
2. The Model Transformation Pattern 7
3. The LML Infrastructure 9
4. Overview of the Eclipse Modelling Project 11
5. The GMF Development Workflow 13
6. The GMF Runtime MVC Architecture 14
7. The Pan-Level Model Meta-Model 16
8. Modelling Modes Overview 17
9. Inversion Example 19
10. Complete, Incomplete, Disjoint and Overlapping Examples 21
11. Intersection Example 22
12. Instantiable Example 23
13. Transitive Connection Example 24
14. Visualization Container Configuration Layers 26
15. Concrete Syntax Ontology and Model 29
16. Concrete Syntax Entity 30
17. Concrete Syntax Connection 31
18. Proximity Indication Example 32
19. Concrete Syntax Feature 33
20. Concrete Syntax Instantiation 34
21. Concrete Syntax Connection 34
22. Concrete Syntax Generalization 36
23. LML Editor Component Diagranm 37
24. LML Editor User Interface 38
25. The Pizza Ontology Example 45
26. The Entity-Relationship Example 46
27. The J2EE UML Profile Example - UML Profile 47
28. The J2EE UML Profile Example - UML Model 47
29. The J2EE UML Profile Example - LML Model 47
30. The Royal & Loyal Example 48
31. Search Order of the Visualizer Search Algorithm 51
32. LML DSL Modelling Mockup 54
33. Extended Organisation Example 56
34. Opening the LML Perspective 66
35. Creating a New Empty Project 67
36. Creating a New Model 67

37. Adding an Ontology to the Diagram 68
38. Adding a Model to an Ontology 69
39. Adding an Entity to a Model 69
40. Adding a Connection to the Diagram 70
41. Connecting Entities with a Connection 70
42. Toggling a Connection 71
43. Showing All Visualizers in a Model 72
44. Selecting a Visualizer 72
45. Editing an Attribute of a Visualizer (Part 1) 73
46. Editing an Attribute of a Visualizer (Part 2) 73
47. The Entity Manipulated by the Visualizer 74
48. Hiding All Visualizers in a Model 74

List of Tables

1. Value Description for the Visualizer's Attributes Trait Key/Value Pairs . . 27

Listings

1. Example of an OCL Constraint 5
2. Definition of the Additional Meta-Model Element in the Ecore PLM Model 39
3. EMF Generator Model Refinement Transformation 39
4. GMF Generator Model Refinement Transformation 41
5. Example for an XPand Template 42
6. The plugin.xml File of the "plm.diagram.custom" Plug-in 43
7. Visualizer Search Algorithm Pseudocode 51
8. XML Serialization of a Model Describing a Shape 52
9. Linguistic and Ontological AllInstances() Operation Example 56
10. Deep AllInstances() Operation and Deep Constraints Example 57
11. Deep Transformation Language Example 57
12. DeepJava Example 59

1. Introduction

Since the release of the Entity-Relationship modelling language in 1976 and the Unified Modeling Language (UML) [26] in the early 1990's no fundamental developments in the concrete syntax of general purpose modelling languages have been made. With today's trends in model-driven technologies and the rising need for domain specific languages the weaknesses of the traditional languages have become more and more obvious. Among these weaknesses are the lack of support for modelling multiple ontological levels or the lack of built-in Domain Specific Language development capabilities. The Level-agnostic Modeling Language (LML) was developed to address these two needs.

In the past a lot of work has been done on the LML and multilevel modelling which motivates this thesis and builds its foundation. The thesis is based on a collection of these papers which are briefly summarized in the following. In [5] Atkinson and Kühne identify the weaknesses of the UML in the domain of multilevel modelling by using a practical example of a small language which is used to describe computer hardware product hierarchies. The paper indicates that a better approach to represent multiple model/instance relationships is needed and proposes the first steps towards such a multilevel modelling language. Later, Atkinson et al. [2] examine the problem from a more theoretical point of view. General problems in current modelling techniques, such as the UML, are identified and workarounds are offered. As a result of this work, the foundation of the multilevel modelling approach was established in the form of the so called Orthogonal Classification Architecture (OCA) [2]. Other papers like [28] and [3] focus on distinct problems when creating a multilevel modelling language. Based on all these observations, Atkinson et al. released an initial description [4] of the so called LML. This paper describes the abstract syntax of the LML and makes suggestions for its concrete syntax. Furthermore, the authors characterise the LML as language which retains the strengths of existing modelling languages and introduces new concepts to overcome their weaknesses. However, some details were left open or are not fully discussed due to space restrictions.

Until now no LML modelling tool has been publicly available to demonstrate the LML in action. Hence, a tool to give other researchers the opportunity to make themselves familiar with the advantages of the LML is needed. As the LML reuses the strengths of traditional modelling techniques this should also be reflected by the implemented modelling environment. The tool's user interface should be comparable with current modelling tools and convenient to use. Through the low learning curve of the new tool's user interface, the acceptance of both the tool and therefore the underlying technology shall be

raised.

The target of this thesis is to completely specify the LML's concrete and abstract syntax. Based on this theoretical work a modelling tool was to be implemented. The tool is implemented using Eclipse and the technologies offered by Eclipse's Model-Driven Software Development (MDSD) ecosystem. By choosing Eclipse as platform a fast adoption of the new technology by the existing community is possible. Additionally, employing Eclipse for the tool offers the look and feel that many modellers are used to. Through the extensible and loosely coupled plug-in architecture of Eclipse, the tool is also capable of building the foundation for future developments of the LML. Such developments include a reasoning service, advanced Domain Specific Language engineering support and the provision of a transformation and constraint language.

The thesis is structured as follows: Chapter 2 describes the foundations of the thesis. It starts by describing the fundamental concepts of Model-Driven Software Development and multilevel modelling. Afterwards, the technologies which are utilized to implement the editor are briefly outlined. Chapter 3 firstly presents the abstract syntax of the LML. Secondly, the concrete syntax of the LML is specified. Chapter 4 discusses the core implementation aspects, such as the LML editor's software architecture. The thesis closes with proposals for future work, related work and the conclusion.

2. Foundations

This section first outlines and describes the theoretical foundations of Model-Driven Software Development, which is the key discipline used in this work. The second part introduces the technologies which are employed to implement the LML editor. These technologies are based on the theories described in the first part.

2.1. Model-Driven Software Development

This chapter introduces the basic concepts of Model-Driven Software Development, which is also referred to as Model-Driven Development (MDD) in the literature. It starts with a discussion of different definitions of MDSD and how the term is used when elaborating on MDSD throughout this thesis. After discussing the term MDSD, the three most relevant MDSD aspects are introduced. These three are meta-modelling, model transformation and Domain Specific Languages (DSL).

2.1.1. Model-Driven Software Development Definitions

Stahl et. al. [44] differentiate between Model-Based and Model-Driven Software Development. Model-Based Software Development (MBSD) is described as "mere documentation, because the relationship between model and software implementation is only intentional but not formal"[44]. In contrast to MBSD, in MDSD "models do not constitute as documentation, but are equal to code"[44]. This distinction shows that for MDSD it is significant that models are the main artefacts of the development process and are not used just for documentation purposes. Furthermore, they mention the Object Management Group's (OMG) Model-Driven Architecture (MDA) Guide as "both a flavour and a standardization initiative for this approach"[44]. This guide defines MDSD as "using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification [of an application]"[23]. Selic [42] defines the key characteristic of MDSD as "software development's primary focus and products are models rather than computer programs" [42]. Additionally, he states that a key promise of MDSD is "that programs are automatically generated from their corresponding models" [42].

All three definitions have in common that they define the role of models in the software development process not as pure documentation but as central artefacts. The last definition by Selic [42] explicitly states that the main concept of MDSD is the full generation of programs out of models which is similar to defining models equal to code [44]. The commonalities of the definitions show that MDSD is a well defined term in literature. These three definitions build the foundation for the term MDSD when used in the following.

2.1.2. Meta-Modelling

The central task when employing MDSD is meta-modelling. The meta-model describes the abstract syntax of a language. It defines the rules to which the language's statements have to conform. Examples for such languages are used in tools for product and software configuration (KobrA [6]), process management (SAP NetWeaver Business Process Management [41]) or tools that describe a certain aspect of a software application (Graphical Modelling Framework (GMF) [9]). For MDSD it is very important that the models are not only for documentation but are equal to code [44] and that the resulting program code is automatically generated out of the meta-models [42]. The generation of a program is done by so called model transformations. A model can be transformed into source code that is understood by a machine or into a model that is interpreted by an engine, e.g. a workflow engine. Chapter 2.1.4 takes a closer look on transformations.

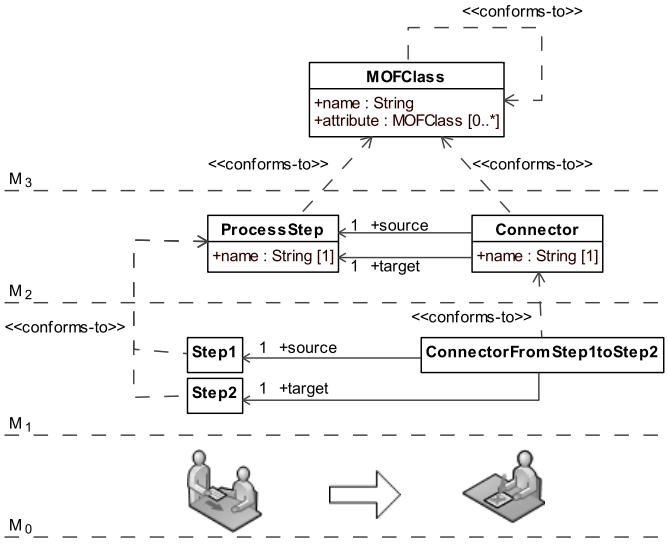

Figure 1: A meta-model example which allows two process steps to be connected with each other through a connection.

Figure 1 displays an example of a small language in the domain of process modelling. The meta meta-model on level M_3 is described by itself. Conforming to the meta meta-model the language's abstract syntax, the meta-model, is described on level M_2. All meta-model elements on M_2 conform to a meta meta-model element on level M_3. At level M_1 an example process is created with elements conforming to meta-model elements on the middle level, M_2. The level M_0, which is displayed beneath M_1, contains the real world instances of the process steps. In this case this could be a real person receiving a contract and checking it in the second step.

To enrich a meta-model "with additional information about the validity of model instances" [44] constraint languages are employed. This information needs to be expressed by constraint languages because it "often cannot be expressed in a diagram" [50]. The most widely known constraint language is the OMG's Object Constraint Language (OCL) [25]. OCL is a modelling language-independent, declarative and side-effect free language [44]. Additionally, it supports design by contract. Design by contract means that an object "is responsible for executing services (the obligations) if and only if certain stipulations (the rights) are fulfilled"[50]. The rights of an object are described by preconditions, the obligations through postconditions. Since UML 2.0, the understanding of OCL moved away from being a pure constraint language to a language for "defining queries, referencing values, or stating conditions and business rules" [50].

```
context ProcessStep
inv: self.name.size() > 0
```

Listing 1: Example of an OCL constraint which forces each process step to have a name.

Listing 1 shows an example of an OCL expression which adds additional validation information to the meta-model at level M_2, displayed in Figure 1. The constraint forces process steps to have names which have more than zero characters. This is also an example of information that cannot be expressed in a diagram by meta-modelling languages like the Meta Object Facility (MOF) [24] or Ecore.

In contrast to the abstract syntax of a language, the concrete syntax describes how the statements in the language visually appear. A language cannot only have a graphical representation, but also a textual one. For instance a UML Use Case Diagram can be represented through a textual notation as well as through a graphical representation. In case of a graphical representation, the concrete syntax specifies that an actor is represented by a stick figure and its name is displayed under it. The textual concrete syntax could specify a two column table for each actor with headings on the left and corresponding information on the right. A real life example for the textual representation of a model is a conditional statement in a programming language such as C# [30]. The conditional statement's concrete syntax specifies that it starts with the keyword "if" followed by a boolean expression in parenthesis and a statement or block that is executed if the boolean statement is true. The abstract syntax only describes that a conditional statement contains a boolean expression and a block of code to execute.

Kleppe [33] states that languages often have multiple concrete syntaxes. Those are "a normal syntax and an interchange syntax" [33]. The normal syntax is the model representation that is displayed to the user while interacting with the model. Even this normal

syntax can have multiple representations, such as a textual and graphical representation for a model. For instance the earlier mentioned UML Use Case Diagram often has a graphical syntax, a textual syntax and an additional XML based interchange syntax. The interchange syntax specifies the model serialization format that is used to interchange models between distinct modelling tools. It often utilizes an Extensible Markup Language (XML) based format, e.g. XML Metadata Interchange (XMI) [29].

2.1.3. Domain Specific Languages

When searching for definitions of the term DSL it is difficult to find a definition that clearly draws a border between General Purpose Languages (GPL) and DSLs. Most definitions give space to argue that a GPL also fits into the definition of a DSL. Ghosh defines a DSL as "targeted at a specific problem" [21] in a problem domain. Kleppe has a similar definition, as she says that a DSL is "describing either a certain aspect of a software system or that system from a particular viewpoint" [33]. These two definitions would include a GPL like C# as a DSL for building applications on the mono platform [38]. Fowler [18] addresses this problem by looking for properties of DSLs that are different to the property of focusing at a specific problem. He says that the significant difference between a GPL and a DSL is the limited expressiveness of DSLs. Therefore, he defines a DSL as "a computer programming language of limited expressiveness on a particular domain"[18]. Limited expressiveness means that a DSL "supports the bare minimum of features needed to support its domain"[18] and that one "cannot build an entire software system in a DSL"[18] but "rather one particular aspect of a system"[18]. When applying this definition to a GPL, such as C#, it becomes clear that such a GPL offers more features than often needed to solve a specific problem and that it can be used to build whole software systems. Employing this definition for the Hypertext Markup Language (HTML), which is a commonly known DSL, shows that HTML can only describe the layout of a web page but cannot describe the look or behaviour of it. Hence, the definition of Fowler enables to unambiguously declare HTML as a DSL, whereas C# needs to be classified as a GPL.

All of the above mentioned definitions speak of a problem domain. The problem domain is the real world where the use case, supported by the DSL, takes place. The DSL provides a user with tools and techniques in the solution domain where the problem is solved. When transferring concepts from the problem domain to the solution domain a vocabulary which is common to both domains is utilized as a kind of glue layer. By using the vocabulary of the problem domain and optional visual metaphors, a DSL is well understood by domain experts.

In literature, DSLs are categorized into external DSLs, internal DSLs, and language workbenches. "An external DSL is a language separated from the language it works with"[18]. This means that a DSL has a custom syntax which is different to the syntax of the language it was implemented in. The expressions of the DSL are parsed into the host language and are then executed. Internal DSLs are "a particular way of using a general-purpose language"[18]. The user is programming with a subset of the host GPL, in which the DSL is implemented in, but experiences the look and feel of a custom DSL. Internal DSLs have the advantage that no parser is needed to translate the DSL expressions into the host language before the statement's execution. "A language workbench is a specialized IDE for defining and building DSLs"[18]. Such language workbenches have the significant advantage that they provide features like a graphical editor, syntax highlighting and code completion for the execution environment at nearly no extra development effort. Languages using a graphical editor instead of a textual one are called Domain Specific Modelling Languages (DSML) by Kleppe [33]. Examples of language workbenches are GMF, used for the LML editor implementation, and MetaEdit+ [48].

2.1.4. Model Transformations

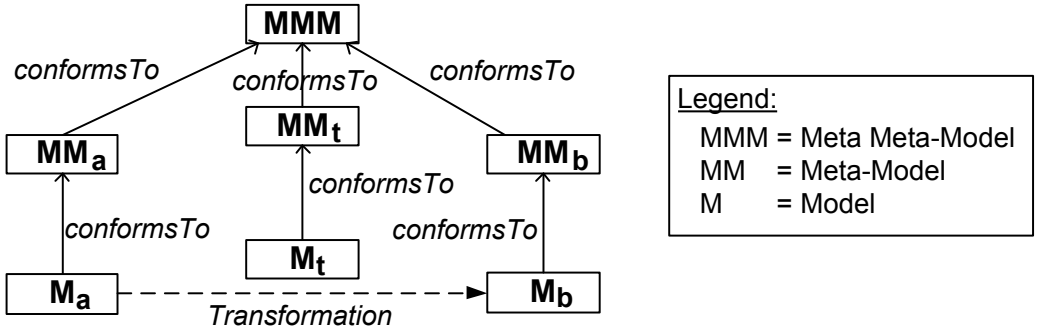

Figure 2: The Model transformation pattern adapted from [14] and [32].

Sendall and Kozaczynski [43] define the term model transformation as "automated processes that take one or more source models as input and produce one or more target models as output, while following a set of transformation rules" [43]. Typical application areas are model synchronization, reverse engineering, view generation, application of patterns and refactoring [43]. Figure 2 shows the relations between the distinct artefacts which take part in the transformation process. It shows a model transformation which creates a target model M_b conforming to meta-model MM_b out of a source model M_a conforming to meta-model MM_a. The mappings between the two meta-models are described on the meta-model level (M_2). Model transformations can be written in a GPL such as C# or with the use of a DSL which is focused on model transformations. A commonly known example for such a DSL is the descriptive and rule based ATLAS Transformation Language (ATL) [32]. In general, two kinds of transformations can be distinguished.

These are Model-to-Model (M2M) and Model-to-Text (M2T) transformations. A M2M transformation takes a model as input and creates a model as output, whereas a M2T transformation produces a textual model as output. A third kind of transformations are Higher-Order-Transformations (HOT) [47], which are a special case of M2M transformations. HOTs are different to the previously mentioned kinds of transformations as they take a model transformation as input and produce a model transformation as output. A use case for such a HOT is the enhancement of existing transformations with functions like tracing a model element through transformations [31]. Due to their special nature of only modifying transformations, HOTs are usually run in the refinement mode. A transformation running in the refinement mode does not create a new model but does an in-place modification of the model.

2.1.5. The Level-agnostic Modeling Language

Current meta-modelling technologies, such as the MOF or Ecore, offer only one type/instance level when modelling: The meta-model at level M_2 and the model at level M_1, which is an instance of the meta-model at M_2. This causes difficulties whenever a problem domain has more than one type/instance level as shown by the example in Figure 3. In [5] Atkinson and Kühne discuss such a case and show the complexity caused by workarounds which are used to overcome this restriction. The LML natively supports modelling over multiple type/instance levels, which are also called ontological levels. Furthermore, the LML clearly separates linguistic from ontological language concerns, which is an "essential difference [... to ...] the traditional four level modelling architecture of the OMG and EMF" [4]. These state-of-the-art modelling technologies mix up linguistic modelling with the ontological modelling of the problem domain. For example, on level M_2 it is specified that an abstract syntax element has a linguistic type (e.g. Class) and a ontological type (e.g. ProcessStep). When creating an instance at M_1, one rather creates an instance of a linguistic element than of an ontological one. Additionally, all constraint and well-formedness checking is performed on the linguistic model elements at level M_2. The LML solves this "asymmetric treatment of ontological and linguistic classification" [2] by clearly separating the linguistic and ontological classification information by employing the OCA with its concept of dual classification [2]. The OCA derives its name from the fact that the linguistic type is orthogonal to the ontological type as shown in Figure 3.

Figure 3 shows an LML diagram with three ontological levels and three linguistic levels. The number of linguistic levels is always fixed to three (L_0, L_1, and L_2), whereas the number of ontological levels (O_0, ..., O_n) is arbitrary. The index in the levels' names is numbered from 0 to n, assigning 0 to the index of the level with the highest degree of abstraction and n to the level with the lowest. When speaking of a higher level, "higher"

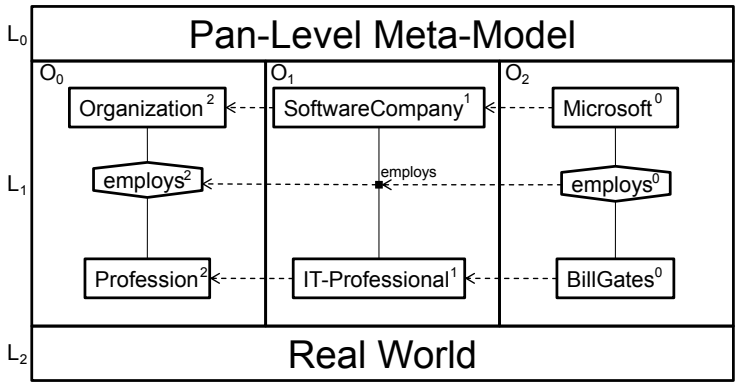

Figure 3: The LML infrastructure adapted from [2]. The example shows a small language that allows to model different types of organizations and their employees.

relates to the degree of abstraction of the compared levels. Hence, a higher level has a lower index. L_1 is the linguistic level on which the LML user defines his model. This linguistic level is split into the ontological levels. The ontological levels and their content are called an ontology. Each model element possesses two types, a linguistic and an ontological type. The linguistic type is determined by the Pan-Level Meta-Model (PLM) that overarches all ontological levels and therefore taken from L_0. The PLM contains linguistic types such as connection and entity. The ontological types are determined by the problem domain and are indicated by "instance-of" relationships. An "instance-of" relationship is represented by a dashed arrow from the instance to its ontological type, which exists on the next higher ontological level. This separation of linguistic and ontological classification allows all constraint and well-formedness checking to be performed on the ontological levels instead of the linguistic ones as in today's tools.

When looking at the middle levels of the UML (M_2, M_1) and the LML (O_1, ..., O_{n-1}), one will notice that the elements on these levels serve as both a type for the elements on the level beneath and as instances for the types at the level above. For example, SoftwareCompany in Figure 3 is instance of Organization and a type for Microsoft at the same time. The LML names this phenomenon the class/object duality. To address this issue the concept called clabjects which originates from the OCA [2] is used. The term clabject is a portmanteau word which combines the terms class and object to show this duality. Together with this new approach the concept of deep instantiation is introduced. Deep instantiation means that to each clabject a potency (clabject potency), to each feature a durability (feature potency) and to each attribute a mutability (value potency) is assigned. This value states how many levels can be influenced by the element. More precisely it defines over how many subsequent levels a clabject can be instantiated or a feature can be passed to a clabject's instances. After each instantiation the value is decreased by one at the instantiated or passed element. Only elements with a value

higher then 0 can be instantiated or passed to an instantiated element. If a value of "$*$" is present, an unlimited number of instances can be created and a feature can be passed over an unlimited number of instantiation levels. A clabject with the potency of 0 is equal to an object as it cannot be instantiated anymore. An attribute with a potency of 0 is equal to the concept of slots in the UML. A more detailed introduction to the topic of potency is given by Atkinson et al. [3]. In Figure 3, nine clabjects can be seen. The three clabjects on the highest level O_0 have the highest potency which is 2. The clabjects on the middle level O_1, which are instances of the clabjects at O_0, have a potency decreased by one. These clabjects at O_1 are both instances of the clabjects at O_0 and types for the clabjects at O_2. The clabjects at O_2 again have a potency decreased by one which now is 0. This means that no instances of clabjects at O_2 can be created.

A DSL, created within the LML, that allows organizations and their employees to be modelled is shown in Figure 3. On the highest level (O_0) the ontology defines that an organization employs a profession. At level O_1 a software company which employs IT-professionals is created as instances of organization, employs and profession. The lowest level models Microsoft as a software company employing Bill Gates. The DSL also allows other types of companies and their employees to be modelled. Creating this DSL with current DSL modelling languages, such as the MOF or Ecore, is not possible without adding additional complexity. These languages would define O_0 as abstract syntax at M_2. M_1 would be occupied by the different types of companies and the types of professions that they employ. To also model the third level, O_2, a modeller needs to apply workarounds, like stereotypes in the UML, to create an additional artificial level.

2.2. DSL Modelling with Eclipse

This chapter gives a brief overview of the technologies on which the LML modelling tool is built. It starts by describing the basis for all components, the Eclipse Rich Client Platform, and then moves on to the distinct Eclipse projects that are utilised for the implementation.

2.2.1. Eclipse Rich Client Platform

A rich client is an application that provides a rich and native user interface (UI) with high speed local processing power. Historically, rich clients replace terminal client applications which have no native UI and do not use local processing power. Today, the number of so called rich internet applications is rising. A rich internet application is an application that runs remotely but has a UI with native operating system metaphors such as drag

and drop. [36]

The Eclipse Rich Client Platform (RCP) is commonly referred to as the "minimal set of plug-ins needed to build a rich client application" [16] with Eclipse look and feel. In the case of the Eclipse RCP these are exactly two plug-ins, org.eclipse.ui and org.eclipse.core.runtime, and their prerequisites [16]. McAffer et al. [36] describe the Eclipse RCP as a portable to multiple operating systems and plug-in based development infrastructure with enhanced deployment capabilities and great development tooling support. The huge ecosystem and availability of plug-ins related to MDSD make the Eclipse RCP a perfect choice to implement an MDSD based tool. Furthermore, modellers can use the Integrated Development Environment (IDE) which they are used to when working with an Eclipse RCP based modelling tool. The next subesctions describe the distinct Eclipse projects that are used to implement the LML editor.

2.2.2. Eclipse Modelling Framework

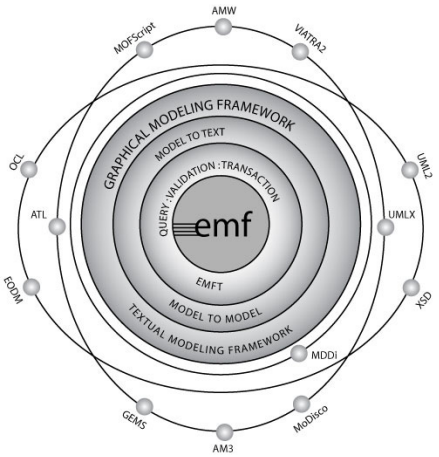

Figure 4: Overview of the Eclipse Modelling Project [22].

Figure 4 gives an overview of the Eclipse Modelling Project (EMP) [11]. The EMP covers technologies for abstract syntax development, concrete syntax development, transformation development, and several MDSD related technologies and research projects [11]. As shown in the figure, the Eclipse Modelling Framework (EMF) is "its core" [45], because "other modelling sub-projects [are] build on top of the EMF core" [45]. EMF itself enables developers to develop applications in an MDSD way by working with meta-models. It offers features such as a generic meta-model editor, Java code generation from EMF models, model serialization, model editor generation, a Java Application Programming Interface (API) to manipulate models, and much more.

The meta-modelling language of EMF is Ecore. Ecore can be compared to the OMG's Essential Meta-Object Facility (EMOF) [24], as EMOF "quite closely resembles Ecore" [45]. EMOF "is the subset of MOF that closely corresponds to the facilities found in [Object Oriented Programing Languages (]OOPLs[)] and XML" [24]. Hence, Ecore can be seen as a subset of the MOF and close to the OMG's MDA standard.

2.2.3. Graphical Editing Framework

The Graphical Editing Framework (GEF) provides an infrastructure to create rich graphical editors and views for the Eclipse platform. The created editors support functions "like drag and drop, copy and paste, and actions invoked from menus and toolbars" [37] out of the box. GEF provides a Model View Controller (MVC) architecture which allows models of various types by a graphical editor to be edited. However, using EMF as model technology provides advantages through EMF's built-in capabilities for model persistence, a notification framework which notifies about model changes and many more [37]. GEF builds the foundations for the editors which are generated by the GMF toolkit.

2.2.4. Graphical Modelling Framework

GMF, which is part of the Graphical Modelling Project (GMP) [12], "was born out of the frustration in creating graphical editors manually (especially in the context of using the Eclipse Modeling Framework)" [1]. Its goal is to enable a tool developer to effectively develop a graphical "What You See Is What You Get" (WYSIWYG) model editor by employing model-driven technologies. GMF is built on top of several Eclipse technologies to achieve this. Those are mainly EMF and GEF. Apart from those two major technologies the Query View Transformation Language operational (QVTo) [27] and Xpand [13] are used for M2M and M2T transformations. Figure 5 and 6 show how the different technologies are related to each other.

The steps and technologies used to create the models that describe the graphical model editor are displayed in Figure 5. The editor is described by four models. These models are the graphical definition model, the tooling definition model, the domain model and the mapping model. The graphical model defines the concrete syntax elements (shapes) that are available in the model editor. The tooling definition model describes the tools available in the model editor's palette. The domain model describes the domain specific language's abstract syntax. The mapping model is used to map a concrete syntax element (shape), and a palette tool to an abstract syntax element. Additionally, the map-

ping model provides the possibility to add validation, creation and value initialisation constraints to mappings. These four models are transformed into a generation model by using QVTo. The generation model serves as an intermediate model during the transformation to the plug-in code. It unifies the four models that describe the editor and offers additional options to configure the generated plug-in. In a last step, the generation model is transformed into an executable Eclipse plug-in. All transformations displayed in Figure 5 can be extended by custom transformations. This gives a developer the ability to extent GMF with features that are not delivered out of the box by employing model-driven technologies.

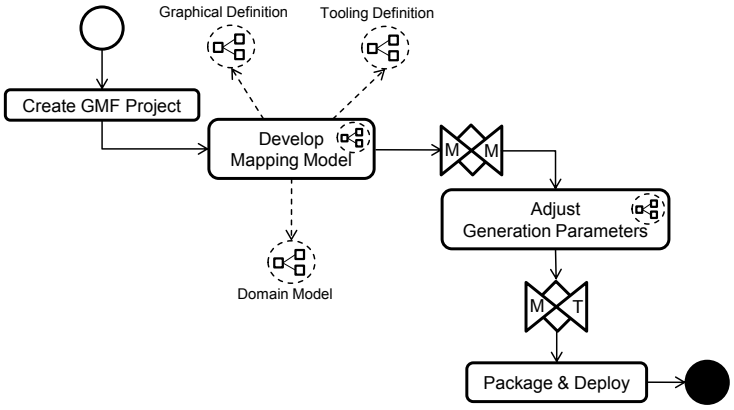

Figure 5: The GMF development workflow from [22].

The deployed plug-in utilizes the GEF and EMF technology to provide model rendering and editing support. Figure 6 displays the interaction of the different application layers. It can easily be seen that the MVC pattern, provided through GEF, is used by the generated plug-in's architecture. The middle layer, the view layer, is used to listen to changes and manipulate the EMF model. When a change to the EMF model is observed GEF updates the controller layer. Changes done in the controller layer are passed to the model layer by GEF and manipulate the EMF model. GMF adds a second tier, the so called notational model, to the view layer. The notational model saves additional information for the model elements like position or visibility. It can be extended to store custom information for model elements, which opens a wide range of possibilities for customization of the generated editor. Besides extending the notational model a developer can extend the behaviour of model elements by adding code and overriding methods in the generated IGraphicalEditParts. To prevent mixing custom code with generated code several extension points are offered by GMF that enable a developer to extend IGraphicalEditParts without touching the generated code. A second way, which is used for the LML editor implementation, is to put the custom Java code into custom Xpand templates that extend the templates which are used to generate the plug-in code

out of the generation model. The templates provided with GMF offer several empty XPand definitions which can be used as a kind of extension points.

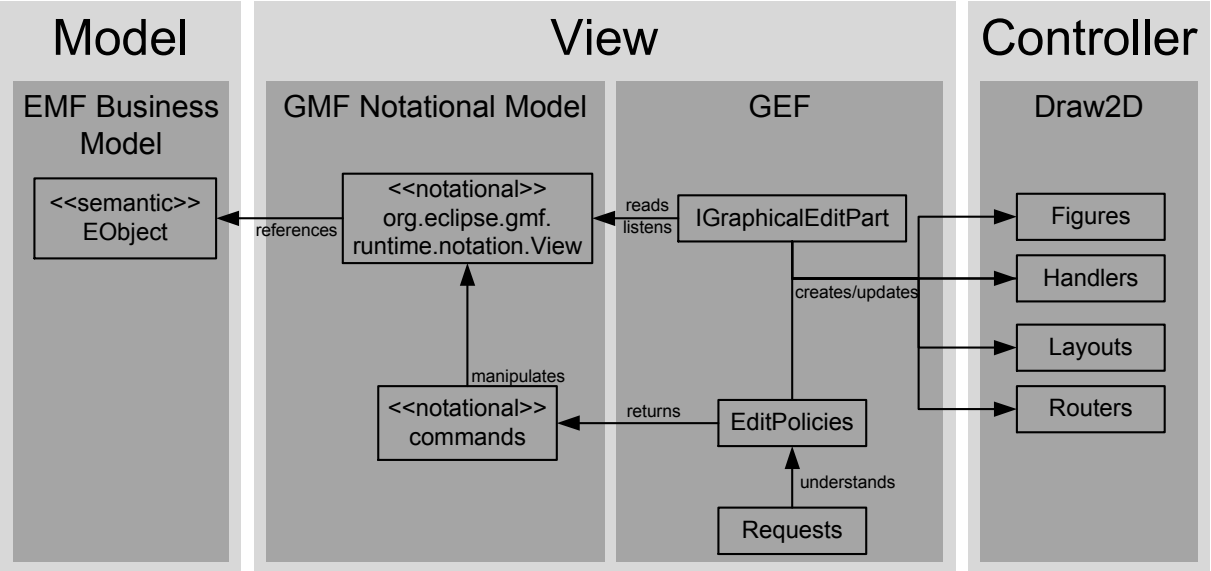

Figure 6: The GMF runtime MVC architecture adapted from [40].

2.2.5. Model Constraint Languages in GMF and EMF

Eclipsepedia gives a comprehensive overview [15] of the usage of constraints in GMF. OCL and other languages are supported to create constraints. Using OCL as constraint language brings the advantage that GMF stores the expressions together with the model in a platform independent way. When using Java, the expressions are stored outside the model and are injected into the generated code at code generation time. For these reasons OCL expressions were chosen for all constraints of the LML editor implementation.

GMF uses constraints for model and link validation. Additionally, it allows meta-model element features to be automatically set-up and meta-model elements to be created for reference features of the created diagram element. Element creation constraints are restricted to only being able to create model elements for containment features of the created diagram element. The element creation and set-up constraints are automatically executed whenever a model element is added to a diagram. Link constraints are evaluated when the modeller draws a connection between two model elements. Only connections that satisfy the constraints can be created. It is possible to define constraints separately for the source and target of a link. Model validation constraints are called audit constraints in GMF. They give the option to the modeller to define validation rules for model elements. These rules provide an error message and a severity (e.g. warning or error) which is displayed

when the rules are violated.

EMF offers the option to enrich a meta-model with additional information through OCL constraints. It offers all types of constraints from the OMG OCL specification [25]. Among these are attribute invariants, pre- and postconditions and the ability to define method bodies. The constraints are defined by annotations which contain the type of the expression as key and the expression itself as value.

2.2.6. Model Transformations in GMF and EMF

The GMF and EMF frameworks make heavy use of M2M and M2T transformations. EMF uses an M2M transformation to generate a generation model out of an Ecore meta-model. Through an M2T transformation Java source code is generated from the EMF generator model. EMF offers the option to extend the M2T transformations by adding custom templates to the transformation process. GMF makes even heavier use of M2M and M2T transformations to foster extensibility. In addition to allowing functionality to be extended by the usage of custom M2T transformations, GMF also supports the execution of custom M2M transformation code. These transformations are executed after the generator model is created. The advantage of using custom transformations for modifying the generated plug-in is that the generated code is not mixed with handwritten code. For this reason all customizations to the LML editor are done through custom M2M and M2T transformations.

3. The Level-agnostic Modeling Language Specification

This chapter gives an overview of the LML's concrete and abstract syntax which is the basis of the LML editor implementation. First the abstract syntax is outlined in detail and then the concrete syntax, which utilizes the abstract one, is illustrated. The thesis follows the description of the concrete and abstract syntax from Atkinson et al. [4].

3.1. Abstract Syntax

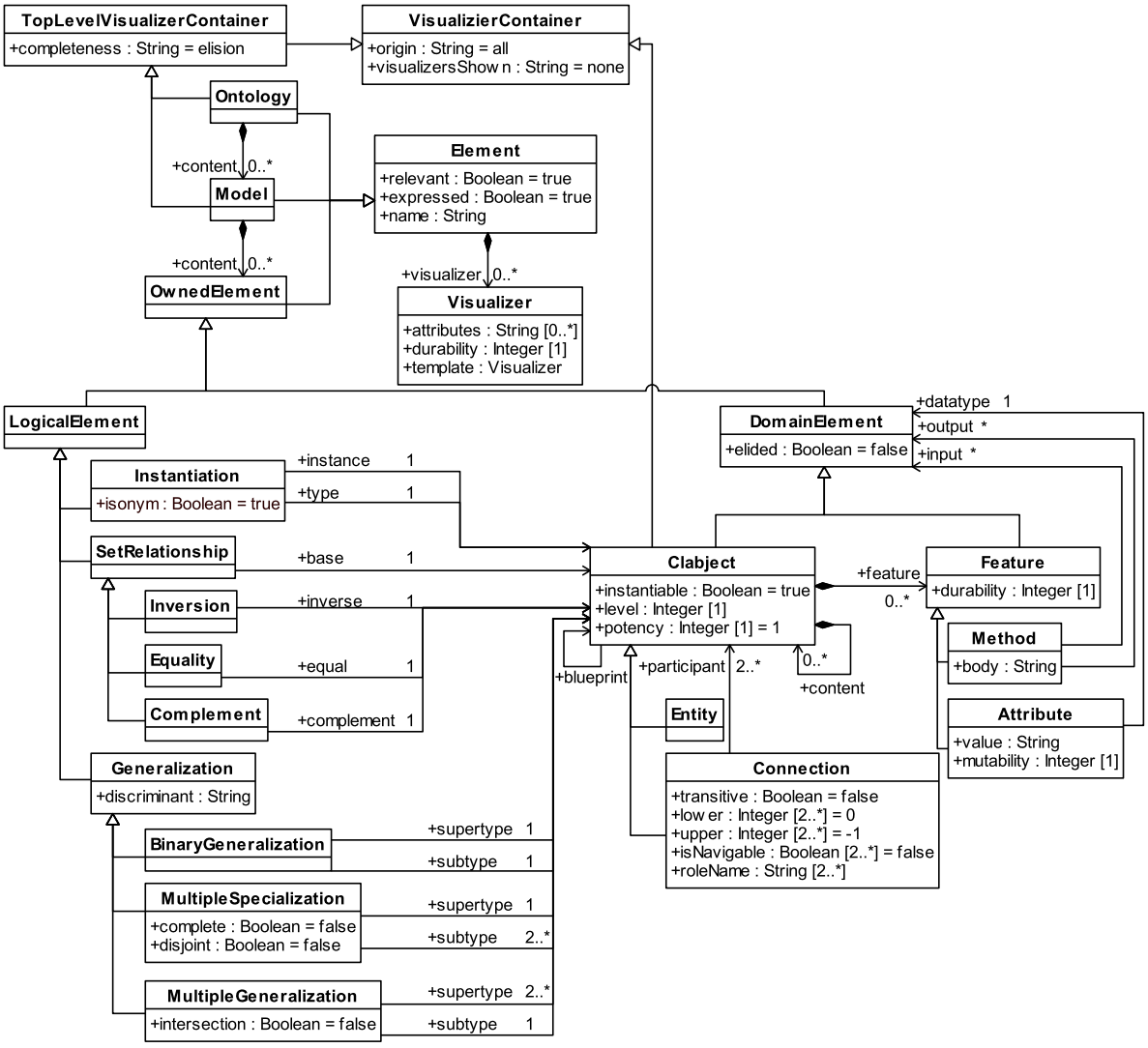

Figure 7: The PLM meta-model represented in the UML.

This chapter specifies the meta-model which defines the abstract syntax elements and their traits used by the LML. The LML's abstract syntax is also called PLM and this name is used in the following when referring to this meta-model. Figure 7 gives an overview of the complete PLM at the time of writing. This is the first released version of the PLM

and can therefore be called version 1.0. Linguistic meta-model element attributes are called traits in order to distinguish them from the attributes which exist on the ontological levels. These were previously called fields. The traits for all meta-model elements are explained below. Traits with a complex meaning get their own paragraph with a detailed description and examples where needed, whereas traits which have a trivial meaning are only briefly introduced. All PLM elements are printed in bold and all traits are printed in italics at the point where their description starts. In this chapter, the names of PLM elements start with a capital letter in order to distinguish them from non meta-model element names.

3.1.1. Modelling Modes

In [3], Atkinson et al. describe the two fundamentally different directions in which a model can be built up. They are called exploratory and constructive mode. In addition, both directions are combined with a scope which specifies whether the created model is a bounded or unbounded model. In a bounded model the number of ontological levels is fixed, whereas in an unbounded model the number of ontological levels is open. However, it is still possible to extend the number of ontological levels of a bounded model if needed. Combining the previous distinctions gives the following four modelling modes: constructive bounded, constructive unbounded, exploratory bounded and exploratory unbounded. The four distinct modelling modes give a slightly different interpretation to the concept of potency introduced below. Three different kinds of potency exist which are clabject potency, durability (feature potency) and mutability (value potency). These differences are discussed later when describing the corresponding abstract syntax elements. All other concepts used in the LML behave in the same way, no matter in which mode they are used. Figure 8 compares constructive and exploratory modelling to each other. It shows that the main difference is the direction in which the ontology is created. Solid elements are created before dashed elements.

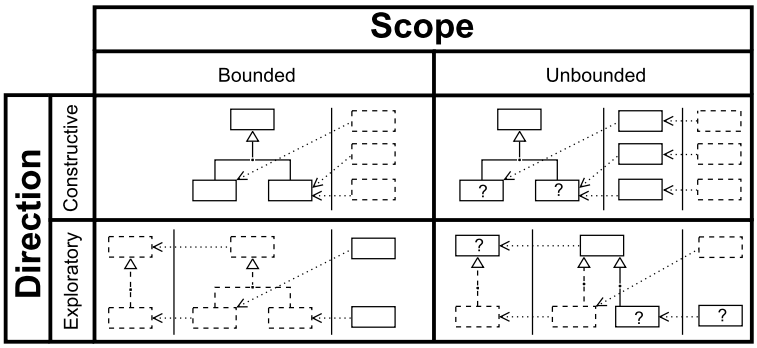

Figure 8: Overview of the available modelling modes adapted from [3].

The constructive mode is the mode in which a software engineer used to model-driven approaches, such as the UML, would develop a model. The modelling starts at the highest ontological level O_0 and instances of the elements on a higher level of abstraction are created on a lower level. As introduced in 2.1.5 the terms higher and lower relate to the ontological level's degree of abstraction and is reflected in the index of the level's name. The lower the index the higher the degree of abstraction. In constructive unbounded mode the number of the lower levels is open, whereas constructive bounded mode fixes the number of ontological levels.

In contrast to the constructive mode, the exploratory mode gives a modeller who is used to ontology technologies, like the Web Ontology Language (OWL) [49], the opportunity to model in a familiar way. It starts at the lowest ontological level O_n, the real world. The real world population is classified and a type is created for each classification on a higher level. This is done until the highest level O_0 is reached. Exploratory unbounded mode is not specific about the number of the higher levels, whereas exploratory bounded mode is explicit about the number of levels.

3.1.2. Element

Element is the root model element from which all PLM core model elements inherit. It possesses three traits which are named relevant, expressed and name. The model elements which do not inherit from Element are concerned with visualization as seen in the PLM meta-model in Figure 7. The *name* trait gives an Element its name which is not unique. Hence, multiple Elements with the same name can co-exist. The *relevant* trait which holds the default value "true" specifies whether an Element is relevant to the modelled domain or not. This value can be changed to "false" by a modeller or reasoning engine, for example for types without instance or Elements without a type. An Element contains one ore more Visualizers in its *visualizer* trait. Visualizers are provided to customize the visualization of model elements. The long term target of the LML's Visualizer concept is to allow the creation of DSLs which use a completely different visualization from the LML's default visualization. This visualizing behaviour is then fully described in the Visualizers and stored in the model itself. At the time of writing, Visualizers are limited to showing traits, hiding traits or adding traits to the Trait Value Specification (TVS). Chapter 3.2 further elaborates on the concrete syntax.

Element.expressed:Boolean: There are two kinds of Elements in a diagram. The ones that are explicitly modelled by a modeller and the others that are computed by a reasoner. All explicitly modelled Elements have expressed set to "true" (default value). The Elements computed by the reasoning engine have expressed set to "false" until the modeller

accepts the computed Elements by setting their expressed value to "true". This feature makes explicit what is suggested by the engine and accepted by the user, which allows reasoning engine assisted modelling in the future.

3.1.3. Ontology, Model and OwnedElement

Ontology is the outermost container of an LML diagram. All Models are contained in an Ontology through the Ontology's *content* trait. **Models** represent the ontological levels of an ontology. A Model contains the modelled solution domain in its *content* trait. **OwnedElement** is the superclass for all elements that can be contained in a Model.

3.1.4. LogicalElement

LogicalElement is the superclass for all model elements which describe the generalization and set theoretic relationships between clabjects. Three flavours of LogicalElements are offered by the LML. These are SetRelationship, Generalization and Instantiation.

SetRelationships show the relations between clabjects based on set theory. They describe the relations between the sets consisting of the clabjects' instances. Possible characterizations of SetRelationships are Equality, Inversion and Complement. They are connected through two traits with the participating clabjects. The *base* trait is the same for all SetRelationships, whereas the other trait is specific to the different types of SetRelationships and is described in the following. SetRelationships are always directed from their base trait to the trait which is named after the SetRelationship type.

Equality.equal:Clabject: The *equal* trait connects the base clabject with the clabject it is equal to. Two clabjects are equal, if they represent the same concept in the problem domain but give a different name to it. For instance the two clabjects flying ant and fertile ant are equal because only fertile ants can fly and only flying ants are fertile.

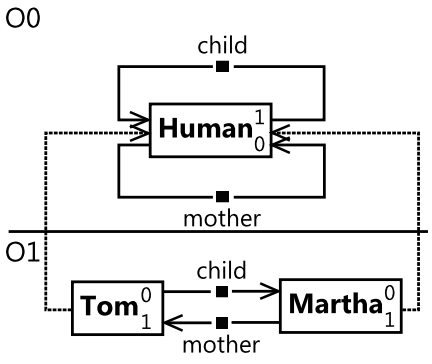

Figure 9: The two inverse connections mother and child.

Inversion.inverse:Clabject: The *inverse* trait connects two connections. Two connections are inverse to each other if they connect the instances of their targets in inverse order. Figure 9 shows the mother connection which is inverse to the child connection. The mother connection connects each mother to its child, whereas the child connection connects each child with its mother. Hence, the two connections connect their target's instances in inverse order.

Complement.complement:Clabject: The *complement* trait indicates that the instances of the base Entity complement the underlying set of instances of the complement Entity. To be able to determine if two Entities complement each other one must have knowledge about the underlying universe's population. If the universe is not given it is assumed as universal. An example for a complement is male which complements female in the universe of all humans.

Instantiations represent a classification relationship between Clabjects. The Clabject stored in the *instance* trait is an instance of the Clabject stored in the *type* trait. In the constructive mode an instance is called an offspring if it is built from the type it is instance of. This type is then called its blueprint. In the exploratory mode being a type means that the type represents a classification built from its instances. In both modes the instances are distinguished between isonym and hyponym. An isonym can be detected by the fact that it has exactly the same properties as its type requires and no more. In contrast, the hyponym has more properties than its type. Whether the instance is an isonym or hyponym is stored in the boolean *isonym* trait. Due to the fact that Instantiations describe classification relationships, they are the only type of relationships that are allowed to cross ontological level boundaries. Instantiations even have to cross ontological level boundaries because only instances of types on higher ontological levels can be created.

Generalization relationships describe the inheritance hierarchy between Entities. Multiple inheritance is provided through the **MultipleGeneralization** relationship which allows a subclass to have multiple superclasses. Having multiple subclasses for one superclass is supported through the **MultipleSpecialization**. A **BinaryGeneralization** relationship which allows one subclass to have one superclass is also offered. In general, for all of these types of generalizations the same generalization meta-model element with a variable number of sub- and superclasses could be employed. However, the PLM does this separation to capture closer information on the generalization and make it accessible to the reasoning engine. The Generalization's super- and subclasses are stored in the traits which are named *supertype* and *subtype*. All three types of Generalizations have

the trait *discriminant*, which gives a name to the Generalization.

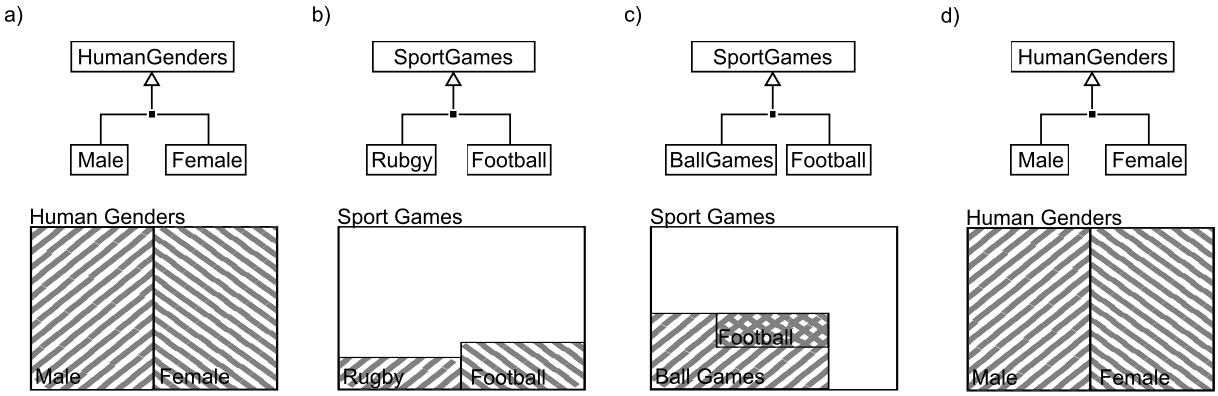

Figure 10: a) complete, b) incomplete, c) overlapping, d) disjoint generalization set

MultipleSpecialization.complete:Boolean = false: The *complete* trait describes the sets of instances defined by the specialization's subclasses in terms of their completeness. The value "true" indicates that all instances of the superclass are also instance of one or more of its subclasses. "False" means that instances of the superclass exist which are not instance of any of the subclasses. An example for "true" is human as superclass with male and female as subclass (Figure 10a). Only instances of human that are either male or female exist. An example for an incomplete set (complete = "false") is the superclass sports with the subclasses football and rugby (Figure 10b). Certainly instances of sports exist which are neither rugby nor football.

MultipleSpecialization.disjoint:Boolean = false: The *disjoint* trait describes the sets of instances defined by the specialization's subclasses in terms of their overlapping properties. "True" (disjoint) means that no instances of the superclass exist which are instance of more than one of the subclasses. On the other hand, "false" (overlapping) indicates that instances of the superclass can be instances of one or more of the subclasses. An example for an overlapping set is sports as superclass with ball games and football as subclasses (Figure 10c). As football is also a ball game the sets are overlapping. An example for disjoint is the previously illustrated example of the superclass human with the subclasses male and female (Figure 10d), because a human can be either male or female, but not both.

MultipleGeneralization.intersection:Boolean = false: The *intersection* trait states that if an instance is an instance of all superclasses it is also an instance of the subclass. Superclasses are allowed to have instances that are not instances of all other superclasses. Instances that are not part of the intersection of all superclasses are not instance of the subclass. An example for an intersection are the superclasses TeamSports and EnduranceSports with their subclass EnduranceTeamSports (Figure 11b). Instances of En-

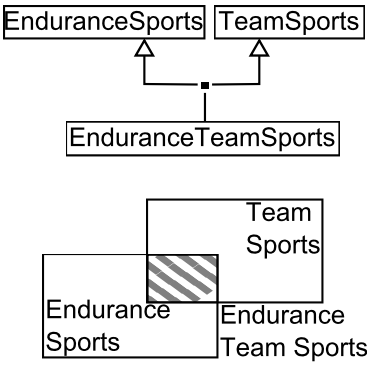

Figure 11: An example for intersection. EnduranceTeamSports is the intersection of its two superclasses EnduranceSports and TeamSports.

duranceTeamSports are sports like football, rugby or handball. These are also instances of the intersection of both superclasses as these sports are team and endurance sports. The superclasses build an intersection because there are existing endurance sports like walking which is obviously not a team sport.

3.1.5. DomainElement

DomainElement is the superclass for all "core modelling elements that fulfil the role of classes, objects, associations, links and features in traditional modelling languages" [4]. Their name, DomainElement, is derived from the fact that they represent entities with their relations and properties in the problem domain. The LML uses Clabjects together with Features to describe these.

3.1.6. Clabject

Clabject is the superclass for Entity and Connection, which represent entities in the problem domain and their relationships. The *level* trait stores the number of the model in which a clabject exists. The level starts with 0 at the model with the highest degree of abstraction, O_0, and is incremented with each following model, e.g. 1 for O_1 and 2 for O_2. Clabjects are characterized by features which are stored in the Clabject's *feature* trait. Additionally, Clabjects can store other Clabjects in their *content* trait.

Clabject.instantiable:Boolean = false: The *instantiable* trait specifies whether a Clabject can be instantiated or not. This can be helpful in cases where a Clabject is intended to be the superclass for other classes but shall not be used for instance creation. This is commonly known as the concept of abstract classes. Figure 12 shows a case where PCHardware is an abstract superclass for Monitor and Motherboard. Hence, it shall not be instantiated on lower levels.

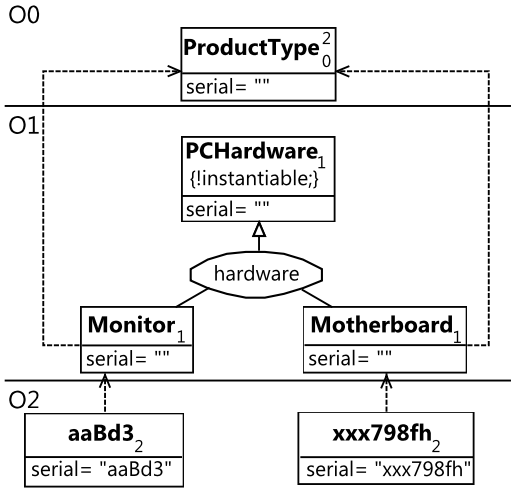

Figure 12: An example for instantiable. PCHardware is the abstract superclass for Monitor and Motherboard.

Clabject.potency:Integer: The *potency* trait specifies over how many subsequent ontological levels the Clabject can be instantiated and therefore influence the levels below it. The potency trait's default value is 1. This corresponds to a UML class, as a Clabject with potency 1 can only get instantiated on the following ontological level. For constructive and exploratory mode the potency has a slightly different meaning. The difference between bounded and unbounded is that for unbounded the potency of "*" is introduced. This value specifies that instances can be created over an unlimited number of subsequent ontological levels. For unbounded models this is needed as the number of ontological levels is left open. The value 0 specifies that no instances can be created. Hence, having a potency of 0 makes a Clabject equal to an object in terms of the UML. In constructive bounded mode the potency is defined as follows. Potency is a non-negative number. An offspring o instantiated from a clabject c must have a potency one lower than c. In constructive unbounded mode, potency can also have the value "*". If an offspring o is instantiated from a clabject c and c has a potency of "*" then o can have "*" or any non-negative number as potency. Exploratory bounded mode defines potency as follows. Potency is a non-negative number. An isonym i of a clabject c must have a potency one lower than c. In exploratory unbounded mode potency can also have the value "*". If an isonym i is instance of a clabject c and c has a potency of "*" then i can have "*" or any non-negative number as potency. "*" is represented in the model by the integer value -1 because the potency is of type integer.

Clabject.blueprint:Clabject: In constructive mode, the *blueprint* is the Clabject from which a Clabject is built during instantiation. A Clabject loses the blueprint value if it is modified in terms of its properties.

The **Entity** meta-model element represents concepts of the problem domain. These concepts might be things like a user, a database or a process step.

Connections represent the interdependencies between Entities. Examples of Connections are a "buys" Connection from customer to product or a "plays" Connection from football player to football. Through the feature trait inherited from its direct superclass Clabject, Connections can hold Methods and Attributes. This feature makes them similar to the concepts of association classes in the UML and connections in Entity-Relationship (ER) Diagrams. The Clabjects participating in the connection are referenced in the *participant* trait. All participants have assigned a lower multiplicity, upper multiplicity, role name and navigability which are stored in the collection traits *lower*, *upper*, *roleName* and *isNavigable*. The default values are: lower = 0, upper = "*", roleName = "" and isNavigable = "true". Here again, the value "*" is represented by the value literal -1. The values of the participant's multiplicity, navigability and role name are assigned by their index in the corresponding collection. Hence, the participant at position 0 in the participant collection has the upper value at position 0, the lower value at position 0, the role name at position 0 and the navigable value at position 0 in the collections for these traits. A connection can only be navigated if the isNavigable trait is set to true. The UML convention that a Connection can be navigated into all directions if no participant has its isNavigable value set to "true", is not applied. If the roleName trait is empty, it is implicitly derived from the name of the participating Clabject. If a Clabject A is connected to a Clabject B then B is also called an associate of A and vice versa.

Connection.transitive:Boolean = false: The *transitive* trait describes the transitivity of the Connection's instances. A Connection is transitive if the following is valid for the Connection's instances: $A \to B \to C \Rightarrow A \to C$. An example for such a case comes from the problem domain of humans and their ancestors. If a human's (A) ancestor (B) is an ancestor of an other human (C) then A is also an ancestor of C. Figure 13 shows Wilhelm being an ancestor of Hannah and Hannah being an ancestor of Marie. Hence, due to the transitivity of the ancestor connection, Wilhelm is also an ancestor of Marie.

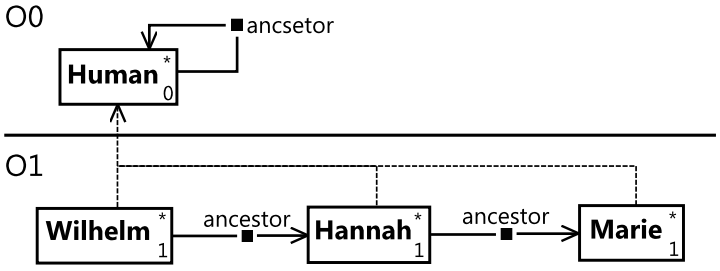

Figure 13: An example for a transitive connection.

3.1.7. Feature

Features enrich Entities by adding Attributes and Methods to them. Attributes store values that describe properties of an Entity like age, price or weight. Methods describe the dynamic behaviour of an Entity. The Feature meta-model element is the superclass for Methods and Attributes.

Feature.durability:Integer: The *durability*, also called feature potency, determines over how many levels a Feature can be passed over to its owner's instances. The default value is the durability of the clabject it is contained in. Additionally, the durability is not allowed to be higher than the potency of its container. A durability of 0 means that a Feature is not handed over to the instances of the Feature's owner, whereas Features with a durability of "*" can be handed over an unlimited number of times. Only clabjects with a durability of "*" can have features with a durability of "*". In constructive bounded mode the offspring o must have a Feature corresponding to each Feature of its type t that has a durability greater than 0. The durability of the feature in o is the durability of the corresponding feature in t decreased by one. In constructive unbounded mode "*" is also available. If the corresponding Feature of its type t has a "*" potency the Feature in o can have a non-negative or "*" potency. In exploratory bounded mode every Feature in the isonym i must have a Feature corresponding to a Feature of its type t that has a durability greater than 0. The durability of the Feature in i is the durability of the feature in t decreased by one. In exploratory unbounded mode "*" is also available. If the corresponding Feature of its type t has a "*" potency the Feature in i can have a non-negative or "*" potency. An Attribute with a durability of 0 corresponds to the concept of slots in the UML. Attributes with a durability of 1 correspond to the concept of attributes in the UML.

The **Attribute** meta-model element contains the *value* trait which holds the default value for the Attribute if the durability is greater than 0. Otherwise it holds the actual value for the attribute. The *datatype* defines which type of data is stored in the Attribute. No complex datatypes are allowed for attributes, i.e by the user modelled Clabjects.

Feature.mutability:Integer: The *mutability*, also called value potency, defines if and over how many levels an Attribute can be changed. The default value is the durability of the Feature. The values for the mutability follow the same rules as the durability with the exception that the mutability can never be higher than the durability. A mutability of 0 means that the corresponding Attribute in the instance of the Clabject must have the same value as the Attribute in the type.

Methods contain the *body* trait which holds the specification of Methods' dynamic behaviour. The body expression of a Method is programmed in the LML's constraint language. The *input* trait allows parameters to be passed to the Method and the *output* trait enables the Method to return a result after execution.

3.1.8. VisualizationContainer and TopLevelVisualizationContainer

VisualizationContainer and TopLevelVisualizationContainer are not core PLM meta-model elements as they do not inherit from Element. They are superclasses for all Elements that contain other Elements and offer the option to show and hide contained Elements based on their trait values. All visualization options are propagated by a shallow mechanism, i.e. options are only applied to the direct content of the container. Models are only influenced by changes at the Ontology level, Clabjects and LogicalElements at the Model level, and a Clabject's content (Features and other Clabjects) at the Clabject level. This enables a fine grained configuration of the ontology's displayed content. Figure 14 gives an overview of the three configuration layers that are available.

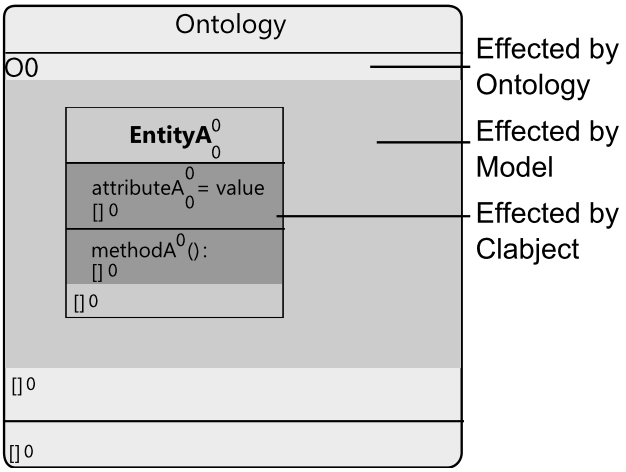

Figure 14: Layers which are influenced by changes to a VisualizationContainer.

TopLevelVisualizationContainer is the superclass for the two outermost containers of an ontology, which are Ontology and Model. It allows elided Elements to be shown and hidden through its *completeness* trait. Possible values for this trait are "elision", the default value, and "noelision". If the value of completeness is set to "elision" all contained Elements which have elided set to "true" are shown as dots. If it is set to "noelision" these Elements are not displayed.

The superclass for TopLevelVisualizerContainer is **VisualizerContainer**. Hence, it is indirectly, via the inheritance hierarchy, the superclass for Ontology and Model. Additionally, Clabject directly inherits from VisualizerContainer. The *origin* trait is used to filter

the visibility of Elements based on their expressed values. The three values “expressed”, “computed” and “all” (default value) are possible. “Expressed” filters out all computed Elements whereas “computed” hides all expressed Elements. “All” adds both computed and expressed Elements to the Ontology. The second trait, *visualizersShown*, determines if the Visualizers contained in the VisualizerContainer are displayed. For this trait two values are possible which are “none” (default value) and “all”. The “none” value hides all Visualizers, the “all” value shows all Visualizers.

3.1.9. Visualizer

Visualizers determine the visualization of the Elements they are contained in. Visualizers build the foundation for the tooling’s advanced DSL capabilities which are introduced as part of the “Future Work” chapter in section 6.3. Similar to Features a Visualizer has a *durability* trait, which determines if a Visualizer is passed to an instance of it’s containing Element. The rules for the Visualizer’s durability are the same as for the Feature’s durability. The Visualizer with the lowest durability is used to visualize an Element if multiple Visualizers are present. The *template* trait supports attribute inheritance between Visualizers.

Visualizer.attributes:String[0..]*: The *attributes* trait is the core concept of Visualizers as this trait stores the information which is used for visualizing the Visualizer’s container. The visualization information is stored in key/value pairs of the form “key= value”. A key/value pair exists for each trait of the containing Element. Possible values for the value of the key/value pairs are “default”, “max”, “tvs” and “noshow”. Table 1 gives an overview and description of these values. Additionally, the Visualizer stores whether a connection is exploded or not. This information is saved for Generalizations, SetRelationShips and Connections. By using key/value pairs a Visualizer can be extended to support more visualization features in the future, e.g. switching visibility of compartments.

Value	Description
default	The default value for all key/value pairs. The trait is shown at the default location defined by the concrete syntax or not at all. For most traits the default location is the shape’s header compartment.
max	Shows the trait with the maximum number of occurrences. For all traits this is the TVS and some are additionally displayed in the header compartment.
tvs	Displays the trait only in the TVS. Traits that are also shown in the header compartment get removed from it.
noshow	The trait is not shown.

Table 1: Value description for the Visualizer’s attributes trait key/value pairs.

3.2. Concrete Syntax

This chapter outlines the LML's concrete syntax. All syntax elements are each described and displayed in a figure. The figures are modelled with the implemented LML editor and are enriched with the concrete syntax element names.

3.2.1. Default Value Handling by the Concrete Syntax

The LML's concrete syntax makes heavy use of the principle of omitting default values when graphically visualizing an abstract syntax element. The traits' default values are shown in the meta-model in Figure 7. However, they are named again when describing the concrete syntax element for an abstract syntax element that possesses a default value. The following paragraphs on the distinct concrete syntax elements explicitly mention whether they must be displayed or can be omitted. The handling of default values can be overridden by the usages of visualizers. In order to do so the value of the key/value pair for the trait to be changed must be set to a value other than "default".

3.2.2. Trait Value Specification

The Trait Value Specification (TVS) is common to all concrete syntax elements. The only elements that do not have a TVS are set relationships, attributes and methods. It is only displayed in case that content is available. When visible it is visualized as curly brackets ("{ }") near the containing element's name. Entries of the TVS are represented in the form "key=value;", where key is the trait name and value its value. The main intention of the TVS is to give a modeller the option to display traits that have no other concrete syntax representation but are valuable information in special situations. Additionally, the TVS is also able to show traits which have a concrete syntax representation. This can be useful if a modeller for example decides not to show a potency next to the name but still wants to show the information. The behaviour for the two presented scenarios can be completely configured via an element's visualizer. Figure 16 (e), 17 (e), 22 (d) and (e) show examples of the TVS in different model elements. Boolean traits which hold the value true are only displayed with their name; if they hold the value false they have "not" or "!" as prefix. No statement can be made about values that are not displayed in the TVS and have no concrete syntax representation.

3.2.3. Visualizer

Visualizers are displayed as square brackets ("[ ]") at the bottom of their container. The visualizer's attribute collection is displayed in the form "key=value", where key is the attribute key and value the attribute value. Only attribute key/value pairs that do not

hold the default value ("default") are displayed. Examples for visualizers are shown in Figure 16 (e), 17 (e) and (f), 22 (h).

3.2.4. Ontology and Model

Figure 15 shows an ontology, which contains three models (O_0, O_1, and O_2). Ontologies are the outermost containers that are displayed in an LML diagram. In the case that the ontology does not contain important information for the current model it is often omitted and only the models are displayed. If an ontology is visible it is visualized as a rounded rectangle with its name in the header compartment and the optional TVS underneath (b). The models which are part of the ontology are added as a stack under the header compartment (a). Models are visualized with their name at the top and the optional TVS on the right beside it (b). The models have a border line at the bottom. Their content is visualized between the name at the top and the bottom line (a). The visualizer of the ontology is shown at the very bottom under the last model (b). The model's visualizers are displayed above their bottom border line (b).

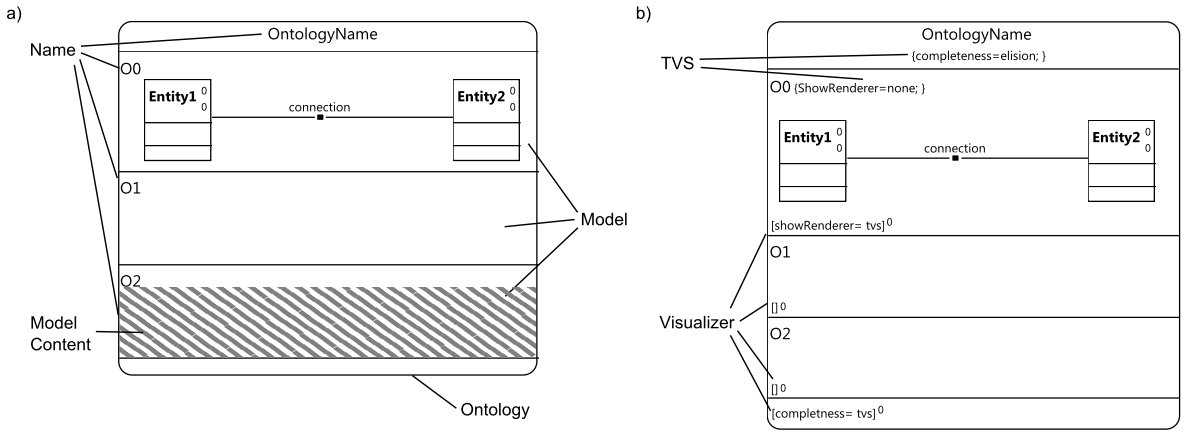

Figure 15: Ontology and model a) without TVS and visualizer, b) with TVS and visualizer.

3.2.5. Entity

An entity is displayed as a rectangle which can contain up to three compartments. The upper compartment, called header compartment, contains the entity's name in bold font. It is the only mandatory compartment of an entity. The name is followed by the entity's potency and level which are not mandatory. Optionally, the TVS can be displayed under the name. Figure 16 (a) shows an entity which has a name, potency and level. In (d) the additional TVS is displayed. The middle compartment stores the attributes, the bottom

one the methods as indicated in a). Computed entities are visualized with a dashed border (b), elided ones are drawn as three dots (e). If an entity is elided all edges from and to the entity are displayed as dashed lines. The visualizer is shown at the very bottom of the entity as in (c) and (d).

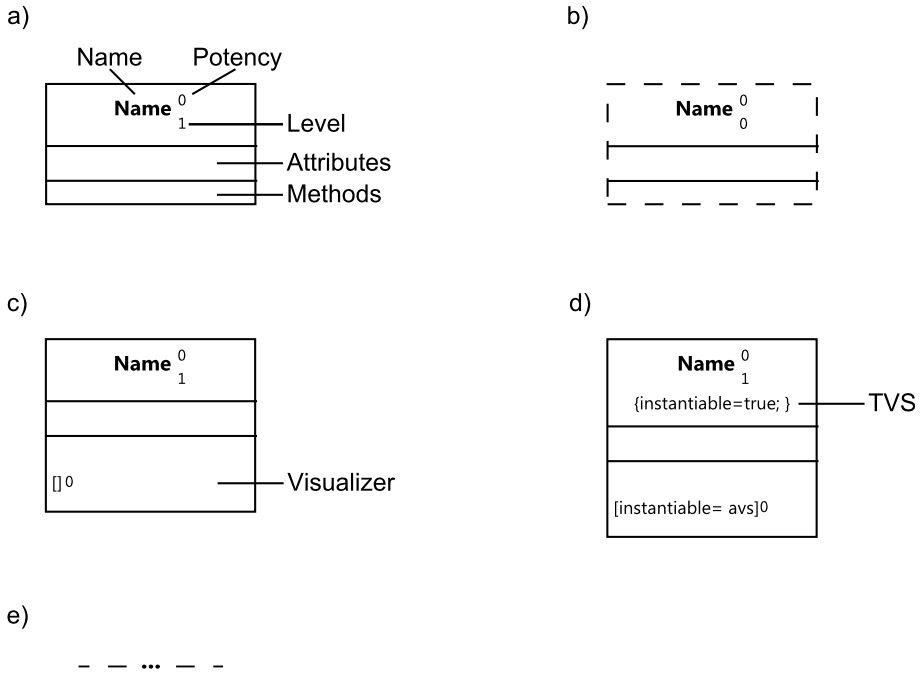

Figure 16: Entity a) without visualizer and TVS, b) computed, c) with visualizer, d) with visualizer and TVS, e) elided

3.2.6. Connection

In their exploded form connections are visualized using a flattened hexagon with three compartments, as shown in Figure 17. The topmost compartment is the header compartment with the name followed by the potency and level (a). If visible, the TVS is displayed beneath the name (e). The middle compartment contains the attributes and the bottom one the methods (a). A connection can also be displayed in a visually insignificant form (c). The visually insignificant form is also referred to as imploded form. When displaying the connection in its imploded form it is usually visualized as a small rectangle with the connection's name at its boundaries. However, all information that is available in the exploded form can be displayed next to the dot. The connection's participants are connected via solid lines (a). The multiplicity and role name is attached to these. The default multiplicity "0..∗" is not displayed. Like entities, connections are visualized with a dashed border if they are computed (b). Visualizers contained in the connection are

displayed at the very bottom of the hexagon (d and e).

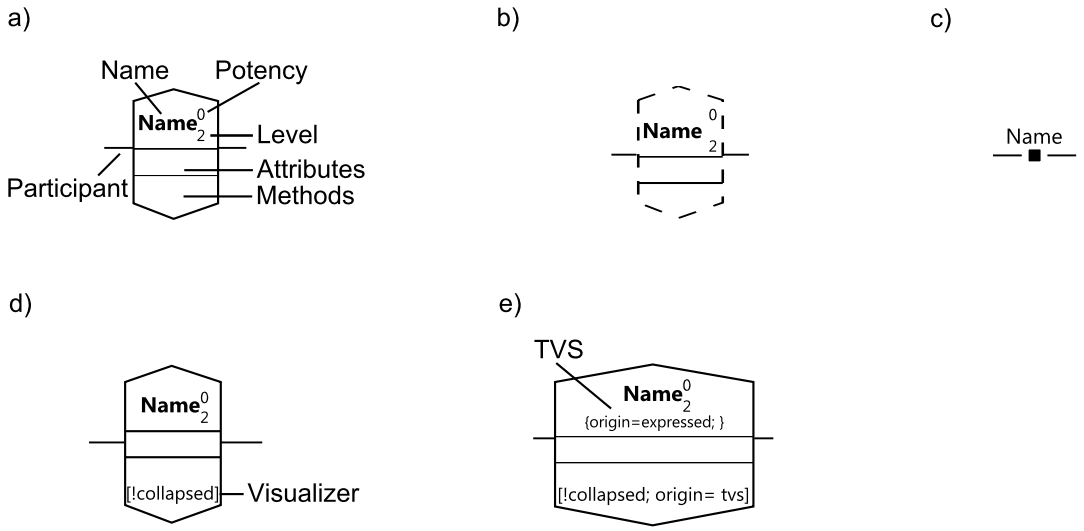

Figure 17: Connection a) without visualizer and TVS, b) computed, c) collapsed, d) with visualizer, e) with visualizer and TVS

3.2.7. Proximity Indication for Clabjects

The concrete syntax element "name" of both connection and entity can support proximity indication. Proximity indication is used to show the location of a clabject within the containment, generalization and instantiation hierarchies. Classification is indicated by "clabjectname:type". This pattern can be applied recursively to display the whole instantiation tree. To omit one or more clabjects in the instantiation tree "::" can be used. Figure 18 (a) shows SoftwareCompany which indicates its "instance of" relationship by its name SoftwareCompany:Organization. Microsoft shows an example for recursively applying ":" and SAP shows an example for elision. Generalizations are indicated by "superclass<clabjectname". Again, this pattern can be applied recursively to display the whole inheritance hierarchy. To omit one or more superclasses "<<" is used. FemaleAnt in Figure 18 (b) displays its superclass by using "<". Furthermore, it is an example for mixing the superclass and classification notation because it also shows that it is of type AntType. Worker is an example for using elision when displaying the inheritance tree. The elision and non elision notation for generalization and instantiation can be mixed as demonstrated by Soldier. Containment is indicated through "container.clabjectname", which is shown by Ant in Figure 18 (b). Ant indicates that it is contained in model O_1 and that this model is contained in an ontology called Antz. This pattern can be used to display the whole containment tree.

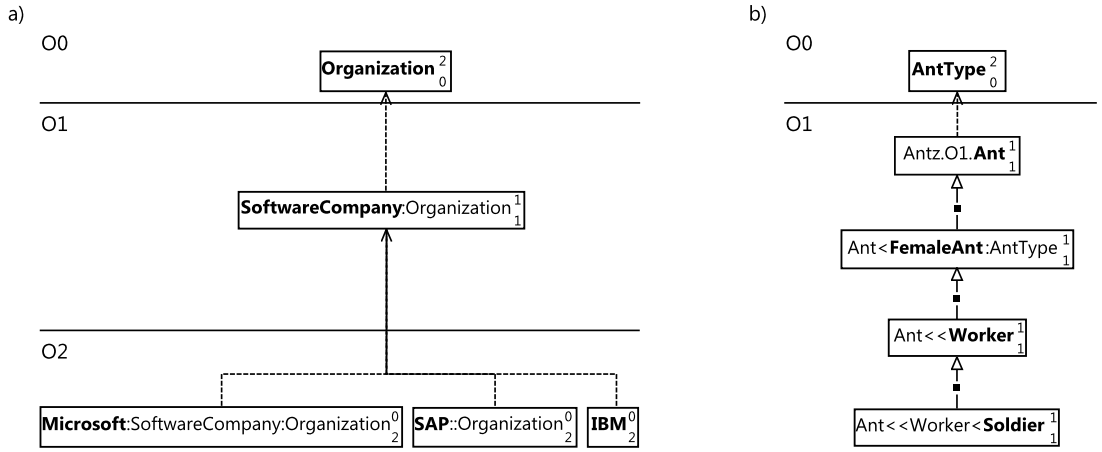

Figure 18: Two examples using proximity indication. a) The previously introduced organization example. b) The Antz example adapted from [4]

3.2.8. Dottability of Relationships

Dottability describes the ability of a relationship to be visualized in two different ways. These are the exploded form and the imploded, visually insignificant, form. The exploded form displays a connection as a flattened hexagon with all of its features. The imploded form hides the features and visualizes a connection as a small black rectangle with usually only its name next to it. However, it is possible to also show all other information except the features in the imploded form. Connections in the imploded form are similar to the connection metaphor in the UML, as the rectangle is this small that the whole connection looks like a single line. For an increased usability of the implemented editor, the imploded connection can be distinguished from the lines that connect it with the participants. Another advantage of using this visualization is that no additional abstract syntax element is needed to support UML association classes or ER like connections. Connections can be displayed in their exploded form to support this scenario. Furthermore, generalizations are visualized by either the imploded form, close to UML, or the exploded form, which allows additional information about the generalization to be displayed. The exploded form of a generalization is visualized by a rectangle with a rounded top and bottom. In the imploded form the dot in the middle of the connection can only be distinguished for usability reasons. Set relationships and instantiations are an exception. The first are not allowed to have an imploded rendering. The latter are the only kind of relationships that do not have the dottability feature. That means the instantiation relationship is visualized as an ordinary line without a visual metaphor in its middle.

3.2.9. Elision

Elision is used to draw the readers attention to special parts of a model by hiding parts that are unimportant in a particular situation. It is possible to elide only entities and features

but not connections. If a model element is elided it is visualized through three dots. Additionally, all outgoing or incoming relationships of an entity are visualized through a dashed line. Set relationships that connect entities are also hidden in the diagram in case that one of the participating entities is elided. The LML allows model elements to be grouped for elision. When doing so, for all grouped model elements only one elision metaphor is displayed instead of a single one for each model element.

3.2.10. Feature

Features are displayed in the corresponding compartment of entities and connections. Figure 19 (a) shows features of an entity, (c) of a connection. Attributes are displayed in the middle compartment with their signature "$name^{duarbility} : datatype = value^{mutability}$". Methods are displayed in the bottom compartment with their signature "$name^{durability}(input) : output$". The only mandatory part of a feature is its name. A feature's visualizer is displayed directly underneath its signature (b and d). Elided features are displayed through three dots instead of their signature. Computed ones have their signature printed in italics. For attributes and methods the default values of their durability and mutability are hidden.

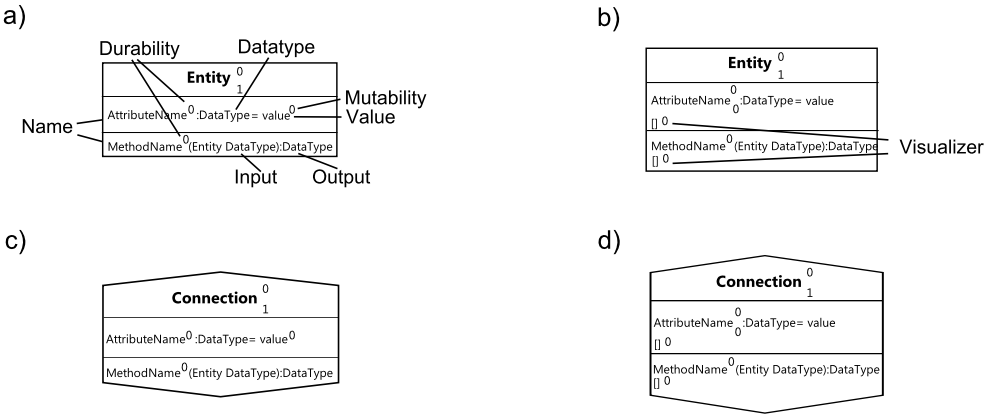

Figure 19: Features a) within an entity without a visualizer, b) within an entity with a visualizer, c) within a connection without a visualizer, d) within a connection with a visualizer

3.2.11. Instantiation

Instantiation relationships are visualized by a dashed arrow directed from the instance to the type (Figure 20). At the endings the kind of instantiation is displayed as text. Possible names for the type ending are: "blueprint" (a), "complete type" (b) and "incomplete type" (c). For the instance ending the possible names are: "offspring" (a), "isonym" (b) and "hyponym" (c). The name is derived from the information of the instance's blueprint

trait (instance.blueprint) and the instantiation's isonym trait. If the instance has a value defined for the blueprint trait and the isonym value is "true", the instantiation is displayed as in (a). If the instance defines a value for blueprint and the isonym value is "false", the instantiation connection is in an invalid state. If the instance's blueprint trait value is undefined and the isonym trait holds the value "true", (b) is used for rendering, otherwise (c) is used. Instantiations are the only kind of relationship that do not have the dottability feature.

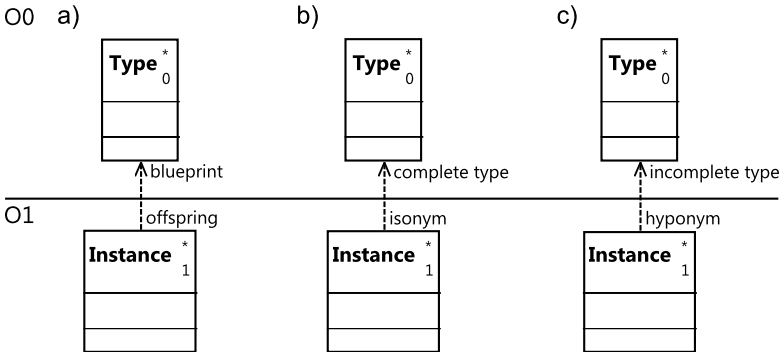

Figure 20: Instantiation with the following values set for instance.blueprint and isonym. a) instance.blueprint <> undefined and isonym = true, b) instance.blueprint = undefined and isonym = true, c) instance.blueprint = undefined and isonym = false

3.2.12. Set Relationship

Figure 21 shows set relationships visualized as rectangles with rounded sides at the top and bottom. They cannot be visualized in a visually insignificant or imploded form. Set relationships do not have a TVS in contrast to all other connections that can be visualized in an exploded form. Furthermore, the three types of set relationships are only distinguished from each other by the name that is displayed in their centre. The different kinds of roles are attached to the connection which represents them. They are "base", "complement", "equal" and "inverse". Visualizations for complement (a), equality (b) and inversion (c) are available. A computed set relationship is visualized by a dashed border (d).

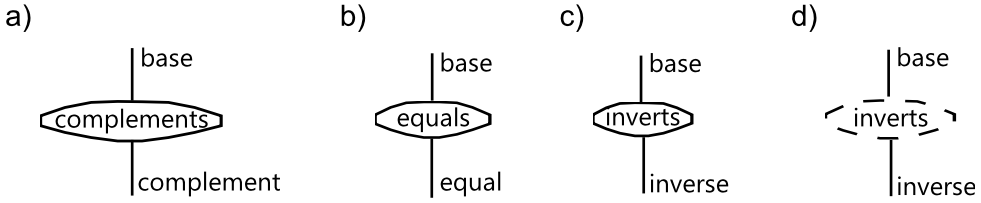

Figure 21: a) complement, b) equality, c) inversion, d) computed inversion

3.2.13. Generalization

Generalizations are visualized with the same shape as set relationships, a rectangle with rounded sides at top and bottom. Centred in the shape the generalization's discriminant is displayed (Figure 22 a, b and d). The TVS is optional and displayed beneath the name when visible (c and e). Superclasses are connected to the generalization through an arrow which has a hollow triangle shape at the superclass end. Subclasses are connected to the generalization through an undecorated line. The only way to visually distinguish the three types of generalizations is by the number of super- and subclasses. The binary generalization connects one sub- and one superclass (d and h). The multiple specialization connects one superclass with two or more subclasses (a, c and f). The multiple generalization connects one subclass with two or more superclasses (b, e and g). Like connections, generalizations can be visualized in an imploded, visually insignificant way. This imploded visualization consists of a small black rectangle with the discriminant displayed at its bounds (f). The generalization's visualizer is displayed at the very bottom of the shape (h). All computed generalizations are visualized with a dashed border (g).

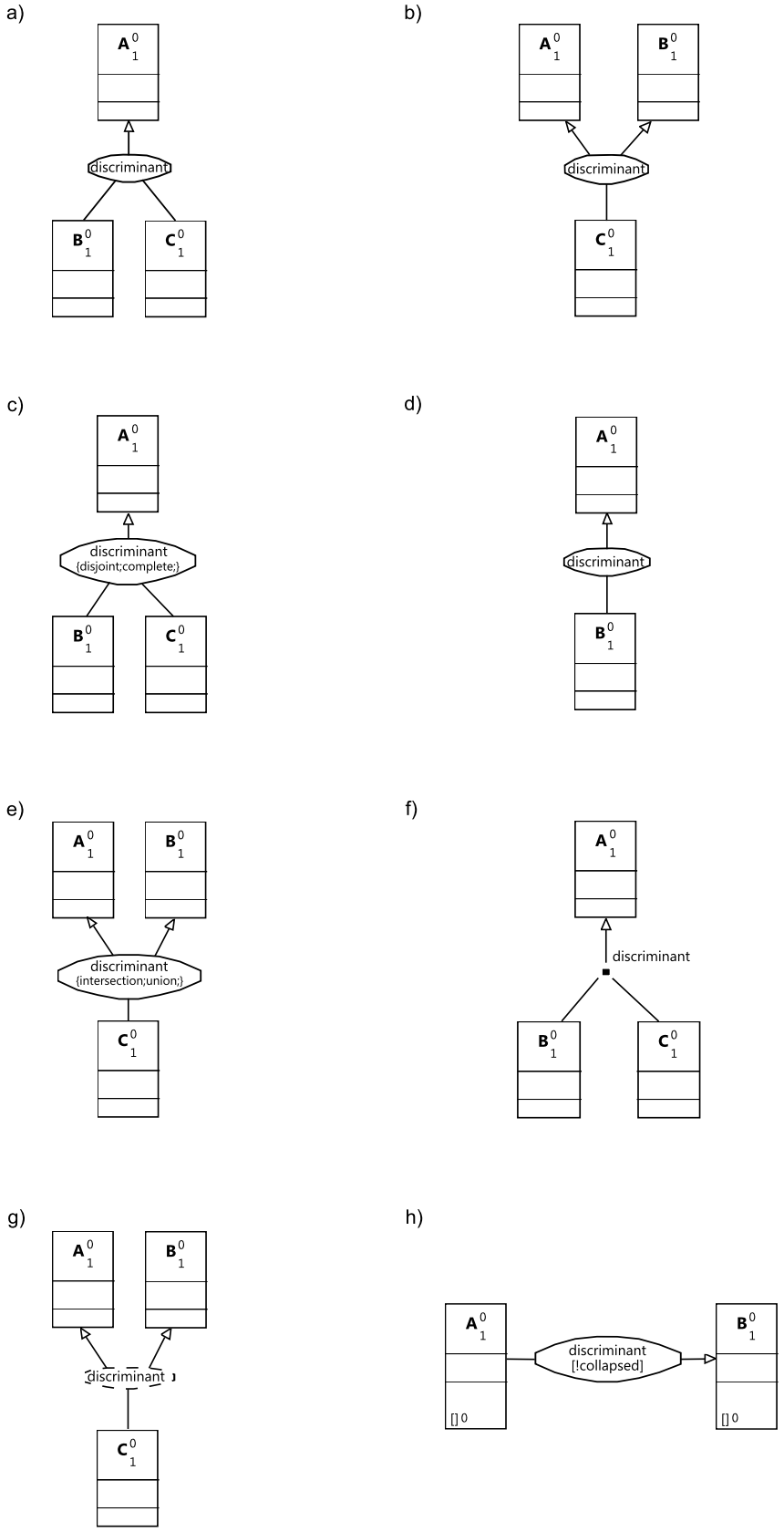

Figure 22: a) Multiple specialization, b) multiple generalization, c) multiple specialization with TVS, d) binary generalization, e) multiple generalization with TVS, f) multiple specialization imploded visualization, g) multiple generalization computed, h) binary generalization with visualizer.

4. The Level-agnostic Modeling Language Editor Implementation

This section first gives an overview of the implemented LML editor and its features, and then it describes how it was implemented. Figure 23 shows the different Eclipse plug-ins represented by components in a UML component diagram. These components are described in the following sections. For better readability the prefix "de.uni_mannheim .informatik.swt", common to all packages, is ommited when talking about plug-in names. The figure also shows a component called "de.itemis.gmf.runtime.extensions", which was not developed as part of this thesis. It was developed within the gmftools project [8]. The plug-in is used for the layout of shapes with an irregular border, such as the exploded form of connections or generalizations.

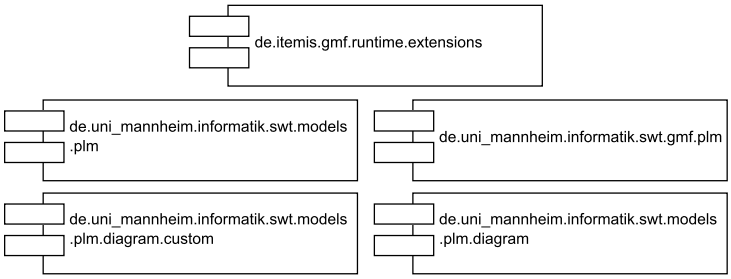

Figure 23: LML editor component diagram.

4.1. The Level-agnostic Modeling Language Editor

The LML editor is provided as an Eclipse plug-in and can be installed into any Eclipse platform. Additionally, it can be deployed as a stand alone Eclipse RCP application. Figure 24 shows the editor's user interface. The editor consists of four parts: the graphical diagram editor in Eclipse's editing area, the properties view at the bottom, the model element palette at the right and the project explorer with the outline at the left. WYSIWG editing is supported by the diagram editor which includes features like seamless zooming, automatic model element alignment and much more. Model elements can be dragged from the model element palette onto the diagram. In the palette, model elements are subsumed into groups which can be expanded or collapsed in order to display only the model elements a user wants to see. Editing of the selected diagram elements is in most cases achieved by clicking on the concrete syntax representation of the trait that is intended to be changed and directly change the value in the diagram editor. For all other cases where directly editing the traits is too complicated to use, e.g. methods and attributes, the properties view is used to change a model element's traits. Changes in the properties

view are immediately reflected in the diagram editor. After changing the selection in the diagram editor all model elements are validated for well-formedness. Model elements on which the validation fails are decorated with a red cross as an error indicator in their upper right. This indicator provides an error description via a tooltip. An overview of the model is provided in three ways. Firstly, the diagram editor enables the user to change the model's zoom level. Secondly, the outline view at the left provides a miniature view of the diagram and a draggable rectangle which indicates the currently viewed part of the model. Thirdly, the project navigator displays the containment tree when a model file is expanded. New model files can be added to all kinds of Eclipse projects. The screenshot shown in Figure 24 displays model files which are added to a general empty project. A detailed user manual is provided in Appendix A.

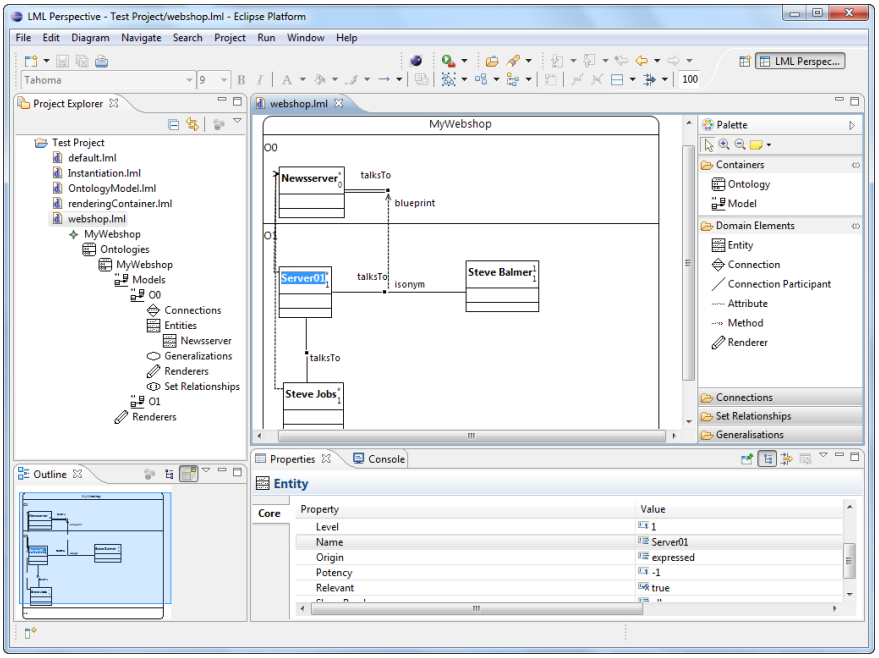

Figure 24: The LML editor user interface.

4.2. Abstract Syntax Implementation

The abstract syntax is defined by using the EMF technology introduced in 2.2.2. The Ecore model corresponds to the PLM, as displayed in Figure 7 and described in 3.1, with the exception that it has one additional outer container added. That container can contain all types of meta-model elements. Hence, the ontology must not necessarily be the root container of a diagram. This is useful if, for example, a modeller decides to only build up a diagram starting from a model or clabject. However, the outer container usually contains the ontology as root meta-model element. Listing 2 displays the additional

meta-model element by using the syntax introduced with the Eclipse project Emfatic [10].

```
class LMLModel {
 attr String name;
 val Element[*] elements;
}
```

Listing 2: Definition of the additional meta-model element in the Ecore PLM model.

Some information that is needed to generate Java code out of the meta-model can only be stored in the EMF generator model which itself is generated by EMF. Thus, an ATL refinement transformation is written which automatically adds this information to the model. This is done because the author of the LML editor is strictly separating generated models from customizations to them. By applying this technique all generated models can be deleted and regenerated without loosing information or having to manually add information to the models. Listing 3 shows the core parts of this transformation. This transformation must be started manually as EMF does not offer an option to automatically run transformations after generating the generator model. The rule "RefineGenFeature" sets the description for each attribute with the help of the "getDescription" helper. This description is displayed in Eclipse's status bar whenever an attribute is selected in the properties view. The rule "RefineGenPackage" adds the package name used for the generarted plug-ins to the model. The rule "RefineGenModel" sets the operationReflection attribute to true which is needed for the OCL support in EMF.

```
module GenmodelRefinement;

create OUT:genmodel refining IN:genmodel;

helper context genmodel!GenFeature def : getDescription : String =
 if (self.ecoreFeature.name = 'name') then
  'Sets the element\'s name.'
 else
  if (self.ecoreFeature.name = 'visualizersShown') then
   --...
  endif
 endif;

-- Adds a properyDescription to the GenFeature
rule RefineGenFeature {
 from s : genmodel!GenFeature
 to o : genmodel!GenFeature (
  propertyDescription <- s.getDescription)
}

-- Sets the basePackage
rule RefineGenPackage{
 from s : genmodel!GenPackage
```

```
  to o : genmodel!GenPackage(
25   basePackage <- 'de.uni_mannheim.informatik.swt.models.plm')
}

-- Sets the basePackage operationReflection
rule RefineGenModel{
30  from s : genmodel!GenModel
  to o : genmodel!GenModel(
   operationReflection <- true)
}
```

Listing 3: The refinement ATL transformation which adds additional information to the EMF generator model.

4.3. Diagram Editor Implementation

The visual editor is modelled in the "gmf.plm" plug-in by using the GMF technology introduced in 2.2.4. The editor uses the concrete LML syntax as defined in 3.2. This chapter focuses on the transformations and custom templates which were used to manipulate the editor generated by GMF. Again, a refinement transformation was chosen to separate the generated GMF generator model from its customizations. Modifications to the editor's generated source code are also needed. These modifications are defined in custom templates that are used to separate custom Java code from the generated Java code.

Listing 4 shows parts of the transformation which is used to enrich the generator model with additional information. It is shortened to focus on the key parts and concepts of the transformation. Parts that are duplicated or similar to previous code are left out which is indicated by three dots ("..."). The key concepts are described in the following paragraphs. In contrast to EMF, GMF offers an automation option to automatically run transformations after the generation of the generator model. As QVTo is the key M2M transformation language used in GMF, automated transformations must also be defined in QVTo. Therefore, QVTo was preferred over ATL.

The most important aspect of the transformation is that it is used as a workaround for two bugs in GMF version 2.3.1. The two bugs are recorded in the bug tracker of Eclipse as "Bug 344104" [20] and "Bug 331875" [19]. The workaround for "Bug 344104" is needed to fix the resizing behaviour of attributes and methods when their visualizers are shown or hidden. Without this modification, the features do not resize to a smaller size after hiding the visualizers. Hence, a massive amount of white space is displayed vertically between the distinct features when visualizers are hidden. This is fixed by Listing 4 lines 9 to 15. A bug concerning nodes with more than one label attached, that use OCL expressions to

calculate their value, is fixed by lines 46 to 49. Without this fix all labels use the same expression to calculate their value and hence display the same value. Lines 22 to 33 set various values for the generated plug-in, such as the provider of the plug-in or whether validation is enabled for the model. The navigation structure used by Eclipse's project explorer is built up in lines 35 to 44. The method "createChildReference" is ommited for space reasons.

```
modeltype GMFGEN uses gmfgen('http://www.eclipse.org/gmf/2009/GenModel');

transformation postRec(inout gmfgenModel : GMFGEN);

property genNavigator : GenNavigator = null;
--...

main() {
 --Needed for Attribute/Method Resize when visualizers are shown/hidden Bug
     344104
 gmfgenModel.objectsOfType(GenChildNode)->asOrderedSet()
  ->select(c | c.editPartClassName.startsWith("Attribute") or ...)->forEach
     (node){
    var defaultSize := new DefaultSizeAttributes();
    defaultSize.height := 22; defaultSize.width := 0;
    node.viewmap.attributes := Sequence{defaultSize}; };

 --Set the list layout for all Attribue, Method etc compartments
 gmfgenModel.objectsOfType(GenCompartment)->asOrderedSet()
  ->select(c | c.editPartClassName.find("Attributes") > 0)->forEach(comp){
   comp.listLayout := true; };
 --...

 --Set up Gen- EditorGenerator, -Diagram, -Plugin and -
     StandardPreferencePage
 genDiagram := gmfgenModel.objectsOfType(GenDiagram)->asOrderedSet()->first
     ();
 genDiagram.validationEnabled := true;
 --...
 this.genEditorGenerator := gmfgenModel.objectsOfType(...)->...->first();
 genEditorGenerator.sameFileForDiagramAndModel := true;
 --...
 this.genPlugIn := gmfgenModel.objectsOfType(GenPlugin)->asOrderedSet()->
     first();
 genPlugIn.provider := "University of Mannheim: Chair for Software
     Engineering";
 --...
 this.genStandardPrefencePage := ...;
 genStandardPrefencePage.name := "LML Diagram Editor";

 --Build up the navigation structure, only Connection is kept for this
     listing
 this.genNavigator := gmfgenModel.objectsOfType(GenNavigator)->...->first()
     ;
```

```
--Group for all Connections
var connectionChildReference := xmap createChildReference("
    ConnectionEditPart", "ModelEditPart", "Connections", "icons/
    connection16.gif");
--Group for all ConnectionAttributes
var connectionAttributeChildReference := xmap createChildReference("
    AttributeEditPart", "ConnectionEditPart", "Attributes", "icons/field16.
    gif");
--...

this.genNavigator.childReferences += OrderedSet{connectionChildReference,
    ... };

--fix BUG 331875
gmfgenModel.objectsOfType(ExpressionLabelParser)->forEach(parser){
  parser.className := parser.className.concat(parser._uses->first().
      container().oclAsType(GenCommonBase).visualID.toString());};
}
```

Listing 4: The transformation refining the GMF generator model.

GMF's M2T templates are extended to manipulate the Java output of the M2T transformations that produce Java code out of the GMF models. In the following, only a short excerpt of code is printed as all modifications to react to attribute changes in the model follow the same pattern. Due to space limitations, the more specific transformations that also override the default visualization etc are not discussed in detail. Listing 5 shows parts of the transformation which is used by all nodes to provide custom behaviour when trait or visualizer values in the model are changed. Lines 3 to 9 show how the "additions" definition is used to provide custom code for an EditPart. Here the "addNotify" method of IGraphicalEditPart is overridden with the method defined in the "addNotify" XPand definition block. The "addNotify" method is used to react to trait and visualizer values when the IGraphicalEditPart is added to the diagram. This method is used for all IGraphicalEditParts when a diagram is opened to set up their visualization. The "handleNotification" method notifies the IGraphicalEditPart about model changes. This method is overridden in lines 11 to 17. Both "AROUND" blocks use if-statements which ask for the EditPart name in order to add custom code only to the EditPart Java class for which it is intended. The same is done for example in the "handleNotificationEventForClabjectLogicElement" definition. This method basically provides the same code for all elements that can be collapsed. Code that differs between the connections is added in an if-statement that checks for the specific EditParts.

```
«IMPORT 'http://www.eclipse.org/gmf/2009/GenModel'»

«AROUND additions FOR gmfgen::GenNode»
  ...
«IF self.editPartClassName = 'EntityEditPart' or ...»
```

```
  «EXPAND setExpressedVisualState-»
  «EXPAND addNotify-»
 «ENDIF»
 «ENDAROUND»

//HANDLE NOTIFICATION
 «AROUND handleNotificationEvent FOR gmfgen::GenNode-»
 «IF self.editPartClassName = 'ConnectionEditPart' or ...»
  «EXPAND handleNotificationEventForClabjectLogicElement-»
 «ELSEIF self.editPartClassName.startsWith('Attribute')»
 ...
 «ENDIF»
 «ENDAROUND»

 «DEFINE handleNotificationEventForClabjectLogicElement FOR gmfgen::GenNode
    »
 /**
  * @generated
  */
  @Override
  protected void handleNotificationEvent(org.eclipse.emf.common.notify.
      Notification notification) {
   super.handleNotificationEvent(notification);

   «IF self.editPartClassName = 'ConnectionEditPart'»
    if (notification.getNotifier() instanceof de.uni_mannheim.informatik.
        swt.models.plm.PLM.Connection)
     updateConnections();
   «ENDIF»
   ...
  }
 «ENDDEFINE»
```

Listing 5: Example for an XPand template.

The plug-in "plm.diagram.custom" is used to implement extension points in order to complement the "plm.diagram" functionality. Listing 6 shows the plug-in's plugin.xml definition. Three different extension points are used. The toggle feature for connections is realized via the "org.eclipse.ui.popupMenus" extension point which allows the context menu to be hooked into for certain object types within Eclipse. Lines 5 to 9 show exemplary the definition for the toggle node feature of connections. To provide an LML modelling perspective "org.eclipse.ui.perspectives" is used (lines 11 to 14). The category for the new diagram wizard is defined via the extension point "org.eclipse.ui.newWizards" (lines 15 to 18).

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
 <extension point="org.eclipse.ui.popupMenus">
```

```
    <objectContribution adaptable="false" id="de...models.plm.PLM.diagram.
        edit.parts.toggleconnectioneditpart" objectClass="de.uni_mannheim.
        informatik.swt.models.plm.PLM.diagram.edit.parts.ConnectionEditPart">
     <action class="de.uni_mannheim.informatik.swt.models.plm.diagram.custom.
         ToggleNodeAction" id="de.uni_mannheim.informatik.swt.models.plm.
         diagram.custom.toggledomainconnectionationaction" label="%action.
         label.0">
     </action>
     <!-- Same for BinaryGeneralizationEditPart, Multiple-
         GeneralizationEditPart, MultipleSpecializationEditPart-->
    </objectContribution>
  </extension>
  <extension point="org.eclipse.ui.perspectives">
   <perspective class="de.uni_mannheim.informatik.swt.models.plm.diagram.
       custom.LMLPerspectiveFactory" id="de.uni_mannheim.informatik.swt.
       models.plm.diagram.custom.perspective1" name="LML Perspective">
   </perspective>
  </extension>
  <extension point="org.eclipse.ui.newWizards">
   <category id="de.uni_mannheim.informatik.swt.lmlcategory" name="LML
       Editing">
   </category>
  </extension>
</plugin>
```

Listing 6: The plugin.xml file of the “plm.diagram.custom” plug-in.

5. Level-agnostic Modeling Language Examples

This chapter gives some examples for LML models which are modelled with the implemented LML editor. The chapter starts with an example that is often used when introducing ontologies. Then it moves on to one of the first available ER diagram examples by Chen. Afterwards, two UML examples are converted into the LML.

5.1. The Pizza Ontology Example

The pizza ontology is used by the Protégé [34] and other tutorials to introduce ontology modelling. The ontology allows different kinds of pizzas and their toppings to be modelled. Figure 25 level O_0 defines types for pizzas and their toppings which both have a price. ToppingType defines an additional attribute weight that states how much topping is used. The connection between PizzaType and ToppingType defines that each pizza has an arbitrary number of toppings. On level O_1 two different instances of PizzaType are created as subclasses of Pizza which are NamedPizza and CustomPizza. Both types of pizza have a fixed price of 3.99. Additionally, the CustomPizza has a calculate price method to add the price of the toppings to the BasePrice of the pizza. Three example toppings are created for combination with a pizza. Those are SalamiTopping, CheeseTopping and MozzarellaTopping. Out of these types on O_1 the PizzaAlberta is created on level O_2. The pizza has 120g of MozzarellaTopping and a price of 3.99. The MozzarellaTopping has no additional price for the pizza. O_2 is the level where all other new pizzas are created.

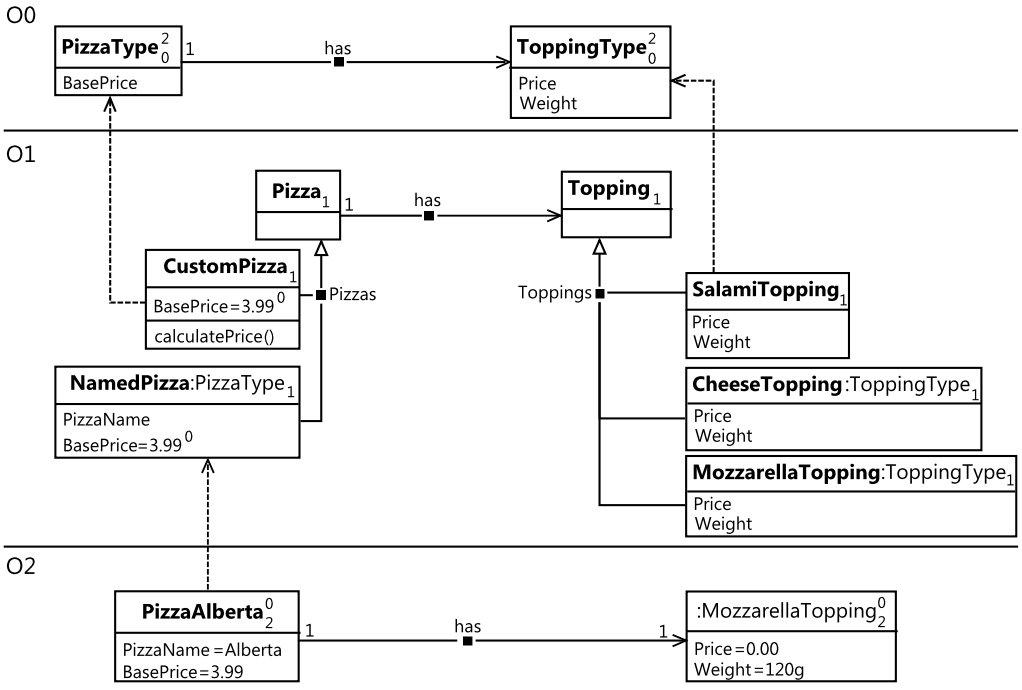

Figure 25: The pizza ontology example.

5.2. The Entity-Relationship Diagram Example

The ER example by Chen [7] models a company consisting of departments, employees, projects and suppliers. Employees belong to departments and work on projects. Each project has a project manager. Parts for the projects are supplied by suppliers. O_0 models the same model as Chen. The O_1 level is added to the example. This level allows to model specific projects with employees working on them. Figure 26 shows a worker of a programming department working on a software product.

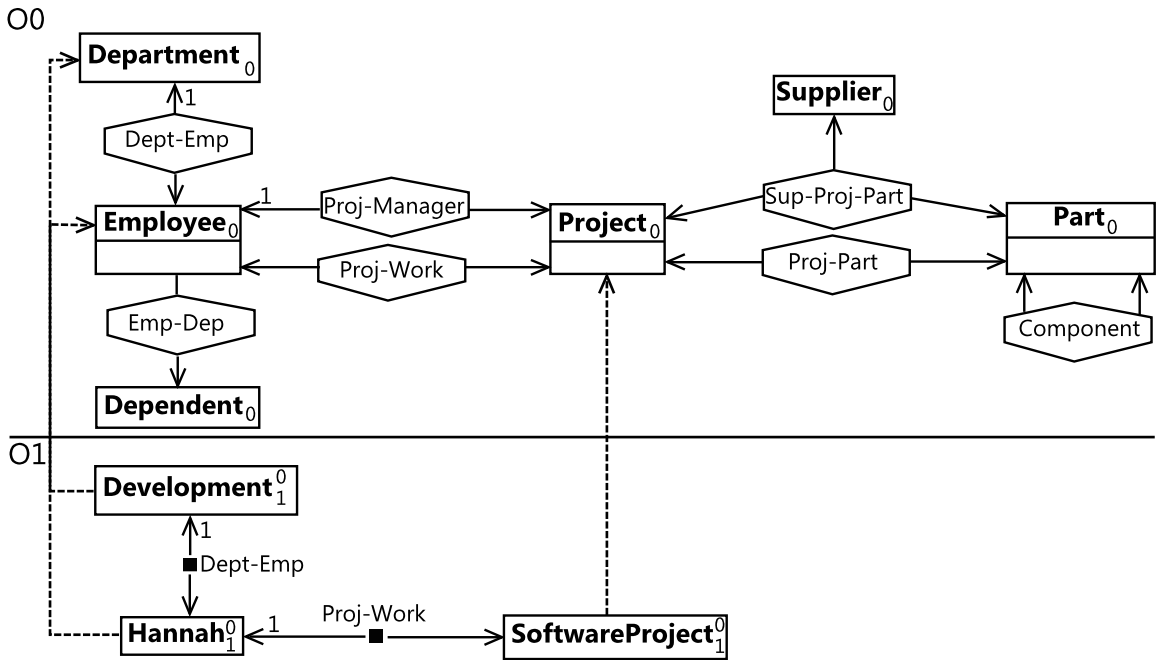

Figure 26: The ER example adapted from [7].

5.3. The Java Enterprise Edition Profile Example

Figure 29 displays the shopping cart of a webshop that is modelled by using the UML profile for Java Enterprise Edition (J2EE). The profile modelled in Figure 29 at level O_0 corresponds to the example from the UML specification [26] (Figure 27). This example shows the strength of the LML in case that more than one type/instance level is present in the problem domain. The UML offers to create only instances of linguistic meta-model elements like connections, classes etc. This profile is used to create an artificial level between the user model and the UML class diagram meta-model in order to introduce new custom types. The UML then uses these types by applying stereotypes to instances of the UML class diagram's linguistic meta-model elements. Figure 28 shows a model which contains a shopping cart with the stereotype session. The UML is not capable to show the profile model (Figure 27) and the user model (Figure 28) side by side in one

diagram. In contrast, the LML natively supports such a scenario. The user defined types are modelled in Figure 29 at O_0 as part of the problem domain and are then used in the model at O_1. Furthermore, the LML supports displaying the two models side by side. The webshop cart modelled at O_1 makes use of the stateful session bean type provided by the profile at O_0.

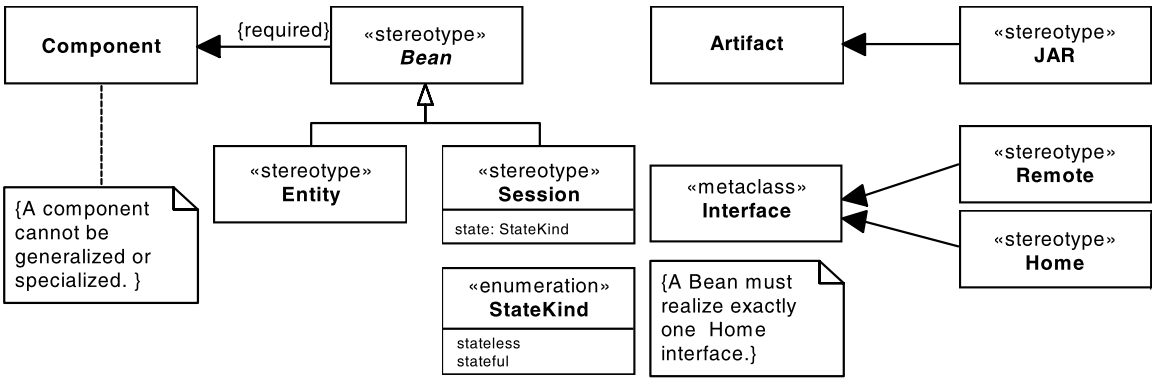

Figure 27: The J2EE UML profile example from [26].

Figure 28: A webshop cart that uses the J2EE UML profile.

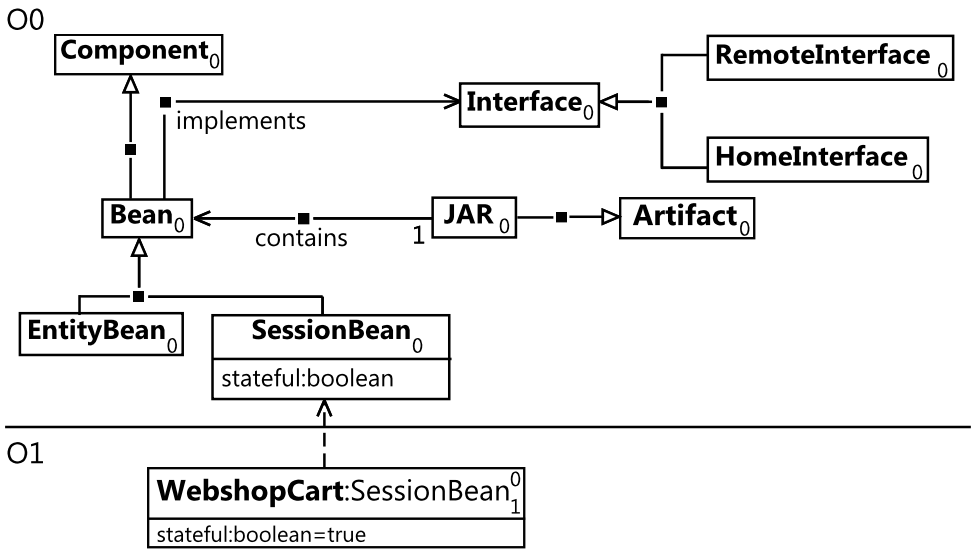

Figure 29: The J2EE UML profile example modelled in LML, adapted from [26].

5.4. The Royal & Loyal OCL Example

The Royal & Loyal example [50] is used by Warmer et al. to illustrate the OCL. Figure 30 shows this example modelled in the LML. The problem domain as presented by Warmer is modelled at O_0. Each customer owns a customer card and takes part in a loyalty program. The programs have different program partners which deliver services. A customer collects points via earning transactions and spends points via burning transactions. Level O_1 shows Marie who is taking part in the "Buy More for Less" partner program.

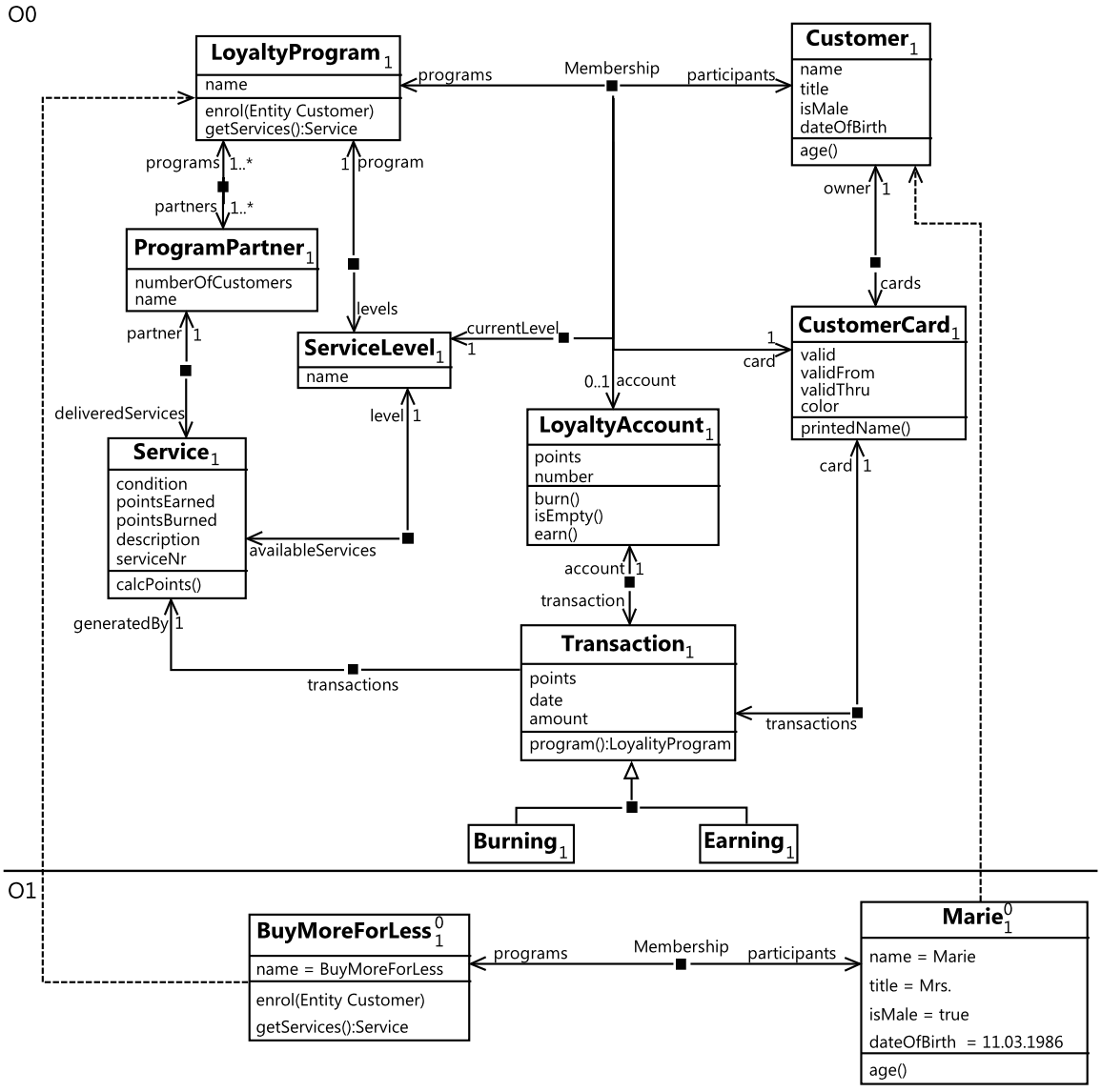

Figure 30: The Royal & Loyal example adapted from [50].

6. Future Work

This chapter gives an overview of the topics that are left open for future work. It starts by describing the limitations of the LML's current abstract and concrete syntax, and the implemented editor. Afterwards, two parts which correspond to the next milestones of the LML follow. They start with the planned future DSL modelling support and finish with a short discussion on the transformation and constraint language. Only initial approaches and research questions are shown in this chapter.

6.1. Abstract and Concrete Syntax

At the time of writing, the PLM has only one known major limitation. Currently, no datatypes for usage with attributes or methods are implemented. The challenge is to enable a modeller to use simple datatypes which are defined in the constraint language side by side with complex user defined datatypes. The user defined datatypes are modelled in an ontology as domain elements. The LML does not allow complex attributes, i.e. attributes with a domain element as datatype. Hence, complex datatypes are only relevant for the in- and output of methods. For methods, a mechanism is missing that allows in- and output variables to be defined which can be conveniently referenced through their name in the method's body expression.

The LML is also missing the capability of marking models as fixed. The target of such a marking mechanism is to warn a modeller who changes a marked model that these changes can break existing transformations, constraints etc. This is very important when shipping ontologies as meta-model for a DSL. Such a mechanism can prevent having many customized dialects of a DSL that are incompatible with each other.

Currently, a conflict exists between the customization of the visualization through visualizers and the default value handling of the LML. In some rare cases one might think of such a customized visualization as the default value handling. This might cause that a reader of an ontology assumes a default value where actually the concrete syntax element which deviates from the default value is hidden. A way to prevent this ambiguity must be found.

6.2. The Level-agnostic Modeling Language Editor

The LML editor implementation allows LML models to be created with a graphical editor. These models follow the LML's abstract syntax, the PLM, and its concrete syntax. The

Scalable Vector Graphics (SVG) export and printing support of GMF produce fair results at the time of writing. The produced images need to be manipulated after exporting to SVG in order to have reasonable results. On big models the automated model validation has performance issues. Here, a solution to do more fine grained validation instead of validating the whole model after a selection change must be found. Furthermore, the instantiation relationships are rather something visual that is restricted by the level and potency of the type than a real instantiation mechanism. A mechanism that allows to create and build instances from types and also validates this relation between type and instance needs to be implemented. To get a look and feel that is more GMF integrated, options like "toggle connection" must be extracted from the context menu and placed on the pop-up buttons offered by GMF. These are displayed next to the model elements when they are selected.

Apart from these implementation issues, the editor lacks much of the functionality that is planned for the LML itself. The reasoning engine which targets at providing assisted modelling and enhanced validation support is not implemented for EMF, yet. Furthermore, the DSL capabilities are not implemented. The transformation and constraint language, one of the key parts of each modelling technology, is also missing. The next two sections give an overview of some enhancements that are planned for the LML and gives a short introduction into these topics. All these features are planned for implementation within the near future to provide an integrated and outstanding modelling experience.

6.3. Domain Specific Language Engineering Support

Deep visualization aims to enable a modeller to build DSLs over multiple levels in a level-agnostic way. In contrast to the state-of-the-art approaches described by Fowler [17], the LML neither separates the concrete syntax definition from the meta-model nor does it need any code generation steps before being able to visualize a DSL. The DSL's concrete syntax is fully deployed with the meta-model and can be dynamically changed at runtime. Visualizers are the key concept of the LML in order to achieve this target. They allow a modeller to store visualization information within the diagram he models. The visualization information is then used at modelling time to manipulate the concrete syntax of the element which contains the visualizer. Visualizers not only determine the concrete syntax of the element they are contained in but also determine the concrete syntax of the instances of the containing element. To enable a separate visualization for model elements and their instances, multiple visualizers are contained by an element. Elements determine which visualizer is used for visualization by looking at the contained visualizers' durability. The visualizer with the lowest durability is used for visualization. Visualizers

with a durability of 0 are only used for visualizing the element they are contained in. All visualizers with a durability greater than 0 are passed to the containing element's instances during instantiation. Hence, these visualizers hold information on how to render the containing element's instances. Again, the instances use the visualizer with the lowest durability for visualization. If no visualizer is present a visualizer is searched for with the search algorithm presented by Atkinson et al. [2].

```
Visualizer findVisualizer()
{
 if (findOntologicVisualizer() != null)
  return findOntologicVisualizer()
 else
  return getLinguisticDefaultVisualizationFor(
      this)
}

Visualizer findOntlogicalVisualizer()
{
 if (this.getVisualizer.size() > 0)
  return this.getVisualizer.get(0);
 else if(getSuperClassesFor(this).size > 0)
 {
  foreach(Element e in getSuperClassesFor(this))
   if (e.visualizer.size > 0)
    return e.getVisualizer.get(0);
 }
 else if(getTypesFor(this) > 0)
  foreach (Element e in getTypesFor(this))
   if (findVisualizer(e) != null)
    return findVisualizer(e);
 else
  return null;
}
```

Listing 7: Pseudocode for the visualizer search algorithm adapted from [2] to support multiple visualizers for an element.

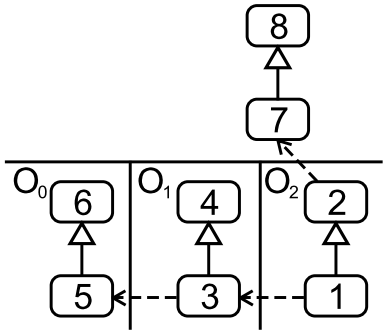

Figure 31: The search order of the visualizer search algorithm from [2].

Listing 7 shows the pseudocode for the visualizer search algorithm. Figure 31 illustrates the algorithm's search order on a brief example. Model elements are represented by rounded rectangles. The order in which these elements are visited by the algorithm is indicated via a number in the rectangle's centre. This algorithm starts searching for a visualizer on the same ontological level as the original element. First, the visualizer is looked up in the element itself (lines 11 to 12). Then, all superclasses are searched (lines 15 to 17). If still no visualizer is found, the ontological types with their superclasses are recursively searched (lines 19 to 22) until the instantiation tree is completely traversed. For the method getSuperClassesFor() it is important to build up the superclass list using

a breadth-first search. This ensures that the closest visualizer in the inheritance hierarchy is found. The getTypesFor() method returns the direct types without traversing the whole instantiation tree. To automatically use the visualizer with the lowest durability it is essential that the visualizer list of all elements is sorted ascending by the visualizer's durability. If the algorithm finds no visualizer, the visualization for the linguistic type is used as a backup (line 6).

To be able to build a DSL the visualizers need to describe the look of the shapes in which they are contained in. In the current implementation visualizers can only influence the visualization by showing or hiding concrete syntax elements. This will be extended by visualizers that can determine the complete look of a shape. In order to create a modelling language which is able to override the concrete syntax of a model element three basic elements are necessary. These are shapes, layouts and labels. By nesting these three types of basic elements in each other powerful DSLs can be created. This is shown by various UI frameworks such as the Windows Presentation Foundation (WPF) and its Extensible Application Markup Language [39] which allows powerful user interfaces to be modelled by nesting controls (here labels), layouts and shapes. Shapes can be predefined standard shapes such as circles, rectangles and rounded rectangles. Also custom shapes can be provided by using coordinate paths and SVG images. The position at which elements are placed inside a shape is determined by the layout. Three layouts, FlowLayout, TableLayout and AbsoluteLayout, already proven in other languages are provided. Labels display model data to the user. They can be directly mapped to ontological attributes or mapped by the usage of a constraint language expression to build more sophisticated values for labels. Listing 8 shows the XML serialization of a model which describes the activity figure with the name "CheckInvoice" found in Figure 32.

```
<RoundedRectangle borderColor="rgb(153,153,153)" backgroundColor="rgb
    (238,238,238)">
 <FlowLayout valign="middle" halign="center" vstretch="center" hstretch="
     center">
  <Label value="name" color="rgb(153,153,153)" />
 </FlowLayout>
</RoundedRectangle>
```

Listing 8: XML serialization of the "CheckInvoice" shape in Figure 32.

Apart from conceptional extensions to the LML, changes to the LML editor's UI must also be made to further support DSL engineering. Figure 32 shows a UI mockup in which the planned changes have already been added. These changes are a second properties view and a second palette. Furthermore, the figure shows the LML's symbiotic language support by displaying the GPL and DSL notation of a diagram side by side in Eclipse's

editing area.

Adding a second properties view is mandatory in order to support the dual classification of model elements. Dual classification means that all model elements have a linguistic and an ontological type which provide traits and attributes to the model element. Traits are determined by an element's linguistic type and displayed in the "Linguistic Properties View". Model elements that are instance of an ontological type have in addition ontological attributes. The GPL notation allows these ontological attributes to be edited by selecting them in the editor and changing the attribute's value trait to the desired value. The DSL notation is not forced to display attribute model elements. Hence, their linguistic value trait cannot be changed via the "Linguistic Properties View". To enable a DSL modeller to change these values by using the DSL notation the second properties view, called "Ontological Properties View", is introduced. This view summarizes all ontological attributes of a model element and offers the look and feel a DSL modeller would expect. The view displays the attributes as a two column table with their name trait on the left and an editing area for their value trait on the right. The editing area allows type sensitive value editing. Boolean values are supported by a true/false drop-down list, text by an one line text box and collections by a dialogue that allows values in the collection to be added, removed and ordered. It is possible for a DSL modeller to model only with the ontological properties view opened. The linguistic properties view can be closed. When doing that a domain expert gets the Eclipse look and feel for editing DSLs and hence does not need to learn any new tools.

The second step on the way to offer an Eclipse integrated DSL look and feel is the context sensitive "DSL Elements" palette. This palette only displays ontological types which can be added to the current level by drag and drop. The linguistic types are completely separated in the "Linguistic Elements" palette. Again a DSL modeller can close the "Linguistic Elements" palette to get the look and feel he is used to when employing Eclipse as DSL tool. This also lowers the learning curve for a domain expert because he at best does not notice that he is not working with a traditional DSL tool but with the LML.

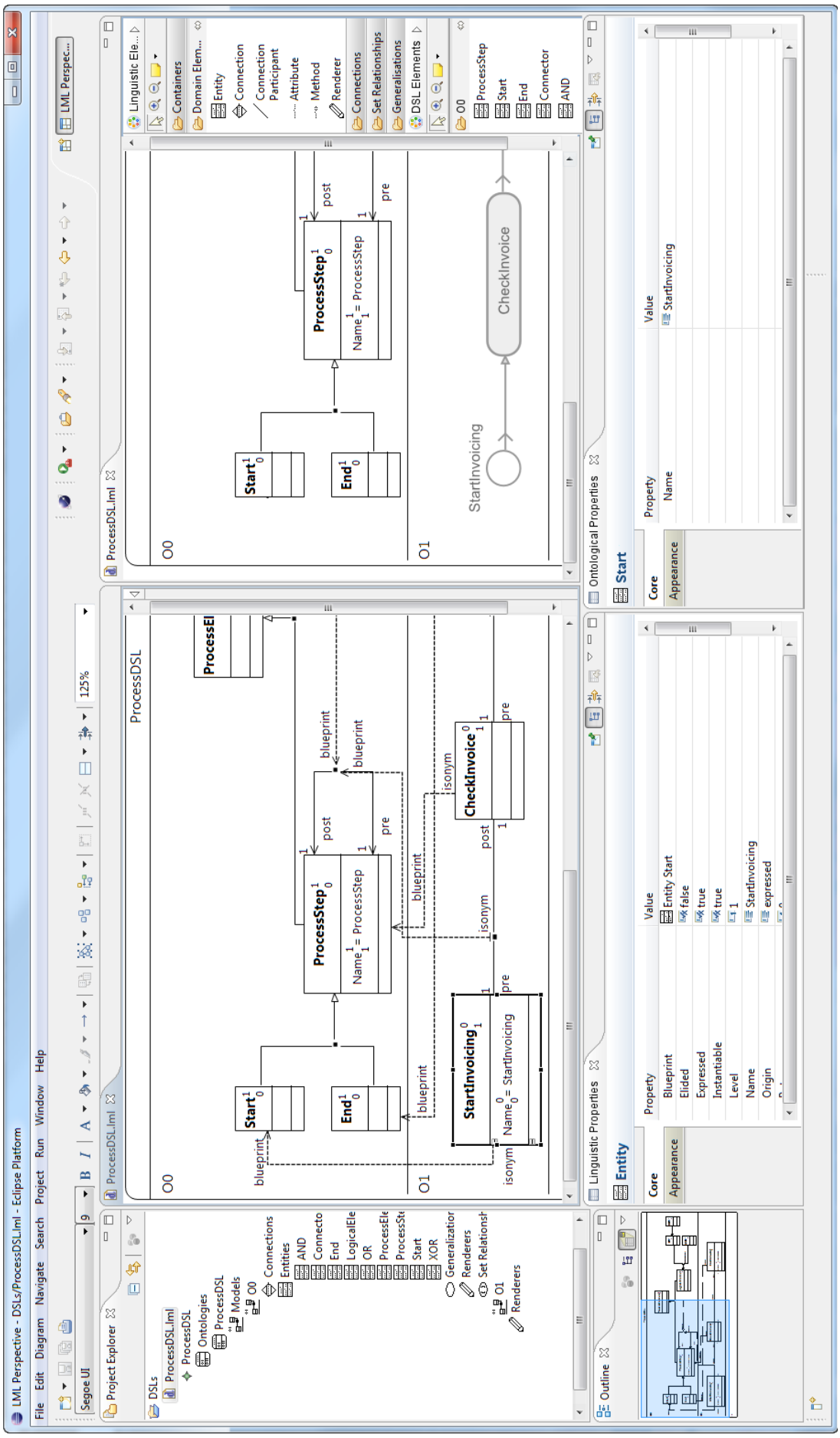

Figure 32: LML DSL modelling mockup.

In the long term more implementation towards DSL modelling in addition to these steps is necessary. A mechanism for hiding distinct levels so that a domain expert only sees the DSL model he works at must be added. Also a mechanism to prevent changes to the levels which define the language meta-model is mandatory. This allows DSLs to be delivered to a wide range of users without the risk that a user uses a customized DSL which can, e.g. break pre-defined transformations or execution engines. A view for defining constraints on the higher ontological levels must be provided to the DSL engineer in order to support model validation that cannot be graphically modelled. Also the concept of visualizers must be extended with an event handling system. Such an event handling system allows DSLs to be implemented with shapes which dynamically react to attribute changes or mouse over events and much more. The events are programmed by using the LML's constraint language. The constraint language mentioned here is introduced in the next section.

6.4. Deep Transformation, Constraint and Query Language

The previous section raises the need for a constraint and transformation language when engineering DSLs. Only a little research on this topic has been done by Atkinson et al. [2]. They suggest a constraint language which is able to support the LML's dual classification and multilevel nature. This is a significant distinction to existing constraint languages such as OCL. Those languages define constraints on the linguistic level M_2 and can influence only one single, following level, which is M_1. The deep constraint language defines constraints on the ontological levels, as these are the ones modelled by the language user. However, at the same time the deep constraint language must also support constraints on the linguistic level L_1. Being able to define constraints at both the linguistic and ontological levels fully reflects the dual classification of model elements. A second requirement is to support constraints that influence more than the direct level below the element on which the constraint is defined. This endows the constraint language with the multilevel nature of the LML. Figure 33 displays the example from 2.1.5 extended by a salary on which the constraint language is demonstrated.

When aiming to support dual classification all OCL operations that access typing information must be redefined or extended. The only operation that is defined in the OCL specification [25] which uses typing information is the allInstances() operation. OCL only defines this operation for linguistic types. Hence, this operation must also be defined to work on ontological types. Depending on the context, either the function for the linguistic or the ontological type is used. Listing 9 lines 1 to 2 show the allInstances() operation in the context of an ontological type. Line 3 shows the operation in context of a linguistic

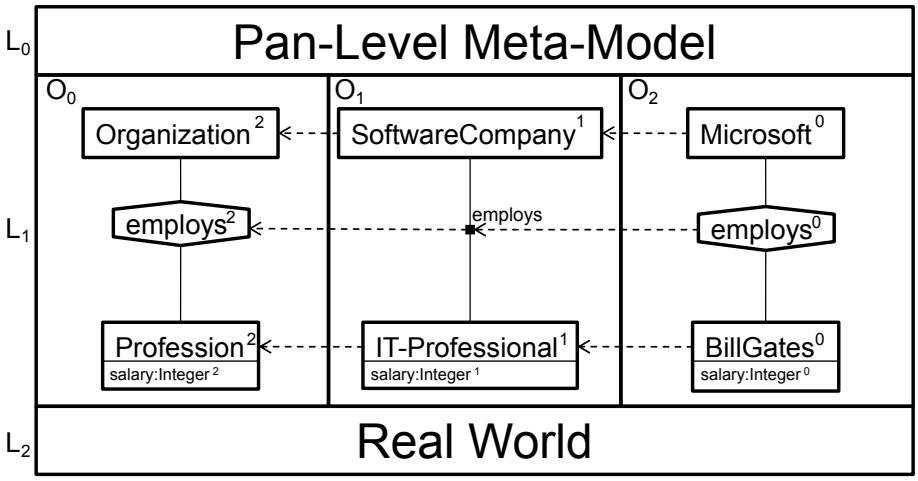

Figure 33: Organisation example.

type. The OCL specification describes a second operation which is called oclIsTypeOf(). This operation's name sounds like it is using typing information but is actually inheritance information instead. This function must be renamed to prevent confusion between inheritance and ontological typing.

```
Profession.allInstances() -- IT-Professional
IT-Professional.allInstances() -- BillGates
Clabject.allInstances() -- {Organization, Profession, Microsoft, ...}
```

Listing 9: Example for the linguistic and ontological allInstances() operation.

For multilevel models a way to navigate to instances of instances etc and constraints that influence more than one level is also needed. For navigation Atkinson et al. suggest to apply the allInstances() operation on the result of the allInstances() operation. In models with multiple ontological levels this can lead to long expressions which are difficult to understand. One needs to count the number of allInstances() operation occurrences to know which level is targeted. Listing 10 line 2 shows an example of an expression which accesses the instances of its instance. Furthermore, it is required to define constraints that are valid for more than the following ontological level. To solve these two problems the widely used concept of potency can be employed. The allInstances() operation can accept a depth as parameter which expresses how deep the operation shall go in the instantiation tree. A parameter with value "*" defines to go to the deepest element in the instantiation tree that has instances. Listing 10 lines 2 to 4 show first an expression using the allInstances() operation recursively and afterwards the allInstances() operation with a parameter. For invariants and pre- and postconditions a potency which describes over how many following levels the condition is valid is introduced. A condition with potency "*" is valid until it is overridden on a following ontological level. Listing 10 lines 6 to 7

define an invariant that forces not only all Professions to have a salary over 30,000 but also its instances to have a salary over 30,000. If applying the invariant of lines 9 to 10, Profession must have a salary over 30,000 but BillGates can have a salary under 30,000. Lines 12 to 13 show an invariant with potency "*" which is applied to all following levels. If a level O_3 would be added, the invariant is evaluated for this level too.

```
IT-Professional.allInstances() -- BillGates
Profession.allInstances().allInstances() -- BillGates
Profession.allInstances(1) -- Bill Gates
Profession.allInstances(*) -- Bill Gates

context Profession
inv(2) minimumSalary: self.salary >= 30,000

context Profession
inv(1) minimumSalary: self.salary >= 30,000

context Profession
inv(*) minimumSalary: self.salary >= 30,000
```

Listing 10: Example for the use of the deep allInstances() operation and deep constraints.

Regarding a multilevel transformation language no publications are available at the time of writing. Hence, only a brief overview of research questions can be given in this paragraph. A rule based transformation language following the example set by ATL can be implemented. This transformation language can also reuse the LML's constraint language, like ATL reuses OCL. By reusing the constraint language only the topics of fitting rules and helpers to the LML's multilevel and dual classification architecture are left open. The dual classification of model elements requires rules and helpers to be defined on ontological and linguistic types. The necessity for defining rules on linguistic types originates from the refactoring perspective. For instance, a modeller wishes to automatically raise the potency of all domain elements by one after appending a model to the bottom of an ontology (Listing 11 lines 1 to 6). The definition of rules on ontological types replaces nearly all existing use cases for transformations. Listing 11 lines 8 to 13 show a use case where a process step is translated to an activity.

```
rule raisePotencyByOne{
 from s : PLM!Clabject
 to o : PLM!Clabject(
  potency <- potency + 1
 )
}

rule processStepToActivity{
 from s : MM1!ProcessStep
```

```
10   to o : MM2!Activity(
       name <- name
     )
   }
```

Listing 11: Example for a deep transformation language.

By supporting the LML's multilevel architecture the question of how a transformation at level n affects the elements of level n+2 emerges. Again, the concept of potency for rules and helpers can be introduced to describe how many following levels are influenced by them. A mechanism to override rules of a higher level on a lower one can be supported, too. However, more extensive research must be done in this field.

7. Related Work

No modelling tool, other than that implemented in this thesis, fully supports the Orthogonal Classification Architecture and LML at the time of writing. The only tool based on the OCA is a research project called DeepJava [46]. DeepJava is a Java dialect that "enables ontological metamodeling" [46] and supports "the concept of deep instantiation and potency" [46]. A drawback of this approach is that, comparing to LML, it is "not intended to support visual modelling or the definition of domain specific languages" [2]. Listing 12 shows a code example which defines a meta-class. It can be observed how similar the LML and DeepJava look like. To some extent DeepJava might be seen as textual concrete syntax for the LML. Therefore, a M2T transformation can be created which translates LML to DeepJava and vice versa.

```
public class ProductType² extends ProductCategory²{

 public ProductType(String categoryName, int categoryCount, int taxRate) {
  super(categoryName, categoryCount);
  taxRate(taxRate);
 }

 int taxRate;

 public void taxRate(int t)
 { taxRate = t; }

 public int taxRate()
 { return taxRate; }

 private float netPrice²;

 public void price(float p)²
 { netPrice = p; }

 public float price()²
 { return netPrice *(1 + type.taxRate / 100f); }
}
```

Listing 12: Example for the definition of a meta-class in DeepJava taken from [35].

8. Conclusion

The LML is motivated by the weaknesses of current modelling technologies. The two identified core weaknesses are the asymmetric treatment of linguistic and ontological typing by technologies such as the UML. Also only a fixed number of modelling levels is supported. This forces a modeller to apply complex workarounds when modelling problem domains with more than one type/instance hierarchy. The LML's architecure and approaches which overcome these weaknesses have been presented. Additionally, the LML's complete abstract and concrete syntax have been consolidated defined. This represents the first complete and publicly available LML specification. Based on this specification the LML modelling tool has been developed. This tool is the first available graphical LML editor that offers the opportunity to experience the LML. Proposals to extend the LML itself and the editor in the domain of DSL modelling have been made. Furthermore, first steps for a transformation and constraint language have been suggested. These are based on the experiences collected during the LML editor implementation. A mockup on how to extend the implemented editor for DSL modelling support has been presented. By completing this work the LML has been made accessible to all modellers.

References

[1] Chris Aniszczyk and Randy Hudson. Create an Eclipse-based application using the Graphical Editing Framework. `https://www.ibm.com/developerworks/library/os-eclipse-gef11/`, download on 1st April 2011.

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

[3] Colin Atkinson, Bastian Kennel, and Björn Goß. Reconciling Constructive and Exploratory Modes of Modelling by Enhancing the Notion of Potency. *Submitted for MODELS 2011*, 2011.

[4] Colin Atkinson, Bastian Kennel, and Björn Goß. The Level-agnostic Modeling Language. In *Proceedings of the Third international conference on Software language engineering*, SLE'10, pages 266–275. Springer-Verlag, 2011.

[5] Colin Atkinson and Thomas Kühne. Reducing Accidental Complexity in Domain Models. *Software and Systems Modeling*, 7:345–359, 2007.

[6] Colin Atkinson and Dietmar Stoll. Orthographic Modeling Environment. In *Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 93–96. Springer Berlin / Heidelberg, 2008.

[7] Peter Pin-Shan Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1:9–36, March 1976.

[8] Dr. Jan Köhnlein. gmftools. `http://code.google.com/p/gmftools/`, download on 26th April 2011.

[9] Eclipse Foundation. Graphical Modeling Framework (GMF) Homepage. `http://www.eclipse.org/gmf`, download on 1st April 2011.

[10] Eclipse Foundation. Emfatic. `http://wiki.eclipse.org/Emfatic`, download on 26th April 2011.

[11] Eclipse Foundation. Eclipse Modeling Project (EMP) Homepage. `http://www.eclipse.org/modeling/`, download on 6th March 2011.

[12] Eclipse Foundation. Graphical Modeling Project (GMP) Homepage. `http://www.eclipse.org/modeling/gmp/`, download on 6th March 2011.

[13] Eclipse Foundation. XPand Homepage. `http://www.eclipse.org/modeling/m2t/?project=xpand`, download on 6th March 2011.

[14] Eclipsepedia. ATL Concepts. http://wiki.eclipse.org/ATL/Concepts, download on 10th March 2011.

[15] Eclipsepedia. GMF Constraints. http://wiki.eclipse.org/GMF_Constraints, download on 27th March 2011.

[16] Eclipsepedia. Rich Client Platform Wiki. http://wiki.eclipse.org/index.php/Rich_Client_Platform, download on 6th March 2011.

[17] Martin Fowler. Language Workbenches: The Killer-App for Domain-specific Languages. http://martinfowler.com/articles/languageWorkbench.html, 2005.

[18] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 1. edition, 2010.

[19] Ralph Gerbig. Bug 331875 - Error When Node Mapping Has Two Expression Labels in gmfmap. https://bugs.eclipse.org/bugs/show_bug.cgi?id=331875, download on 17th May 2011.

[20] Ralph Gerbig. Bug 344104 - DefaultSizeAttribute via gmfgraph. https://bugs.eclipse.org/bugs/show_bug.cgi?id=344104, download on 17th May 2011.

[21] Debasish Ghosh. *DSLs in Action*. Manning Publications, 1. edition, 2010.

[22] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 1. edition, 2009.

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

[24] Object Management Group. Meta Object Facility (MOF) Core Specification. http://www.omg.org/spec/MOF/2.0/PDF/, 2006.

[25] Object Management Group. Object Constraint Language Version 2.2. http://www.omg.org/spec/OCL/2.2, 2010.

[26] Object Management Group. OMG Unified Modeling LanguageTM (OMG UML), Infrastructure. http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/, 2010.

[27] Object Management Group. Query/View/Transformation Specification Version 1.1. http://www.omg.org/spec/QVT/1.1/PDF/, 2011.

[28] Matthias Gutheil, Bastian Kennel, and Colin Atkinson. A Systematic Approach to Connectors in a Multi-level Modeling Environment. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 843–857. Springer-Verlag, 2008.

[29] ISO. *ISO/IEC 19503: XML Metadata Interchange Specification - Version 2.0.1*. ISO (International Organization for Standardization), 1. edition, 2005.

[30] ISO. *ISO/IEC 23270: Information technology – Programming languages – C#*. ISO (International Organization for Standardization), 2. edition, 2006.

[31] Frédéric Jouault. Loosely Coupled Traceability for ATL. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, pages 29 – 37, 2005.

[32] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008.

[33] Anneke Kleppe. *Software Language Engineering: Creating Domain-specific Languages Using Metamodels*. Addison-Wesley, 2009.

[34] Holger Knublauch, Alan Rector, Robert Stevens, Chris Wroe, Simon Jupp, Georgina Moulton, Nick Drummond, and Sebastian Brandt. A Practical Guide To Building OWL Ontologies Using Protégé 4 and CO-ODE Tools Edition 1.3. `http://owl.cs.manchester.ac.uk/tutorials/protegeowltutorial/`, 2011.

[35] Thomas Kühne and Daniel Schreiber. Can Programming be Liberated from the Two-level Style: Multi-level Programming with Deepjava. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 229–244, New York, NY, USA, 2007. ACM.

[36] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*. Addison-Wesley, 2. edition, 2005.

[37] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden. *Eclipse Development Using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM, 1. edition, 2004.

[38] Novell. Mono Project Homepage. `http://mono-project.com/`, download on 10th March 2011.

[39] Charles Petzold. *Applications = Code + Markup: A Guide to the Microsoft Windows Presentation Foundation (Pro - Developer)*. Microsoft Press, Redmond, WA, USA, 2006.

[40] Graphical Modelling Project. Eclipse Help: Developer Guide to Diagram Runtime Framework. `http://help.eclipse.org/helios/index.jsp?topic=/org.`

eclipse.gmf.doc/prog-guide/runtime/DeveloperGuidetoDiagramRuntime.html, download on 6th March 2011.

[41] SAP. Components & Tools of SAP NetWeaver: SAP NetWeaver Business Process Management. http://www.sap.com/platform/netweaver/components/sapnetweaverbpm/index.epx, download on 6th March 2011.

[42] Bran Selic. The Pragmatics of Model-driven Development. *Software, IEEE*, 20(5):19 – 25, 2003.

[43] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-driven Software Development. *Software, IEEE*, 20(5):42 – 45, 2003.

[44] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 1. edition, 2006.

[45] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2. edition, 2009.

[46] Thomas Kühne. Multi-Level Programming with Java. http://homepages.mcs.vuw.ac.nz/~tk/dj/, download on 9th May 2011.

[47] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. *Lecture Notes in Computer Science*, chapter On the Use of Higher-Order Model Transformations, pages 18–33. Springer Berlin / Heidelberg, 2009.

[48] Juha-Pekka Tolvanen. MetaEdit+: Domain-specific Modeling for Full Code Generation Demonstrated [GPCE]. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 39–40, New York, NY, USA, 2004. ACM.

[49] W3C. Web Ontology Language (OWL). http://www.w3.org/2004/OWL/, download on 4th April 2011.

[50] Jos Warmer and Anneke Kleppe. *Object Constraint Language 2.0*. mitp-Verlag, 2004.

A. LML Editor User Manual

This appendix provides a user manual for the editior implemented in this thesis. It starts by describing the editor and then proceeds to show how to use the editor.

A.1. Installation

The editor can be downloaded with a complete Eclipse distribution as a 32bit or 64bit version. Optionally, it can be downloaded as a plug-in which can be installed into any running Eclipse distribution. The plug-in provides all needed dependencies. When downloading the complete Eclipse package, the zip file must be extracted to the hard drive. No additional steps are required. For Windows systems it is important to extract the files to the root of the partition, e.g. "c:\eclipse\", to prevent path names exceeding the maximum number of characters allowed by Windows. The plug-in can be extracted into the dropins folder of any Eclipse installation. For this scenario the folder structure "\dropins\lml\eclipse\plugins\" is recommended.

A.2. Walkthrough: Creating a Diagram File

Open the LML perspective in Eclipse in case this is not opened yet. Click on the "Open Perspective" button (Figure 34, 1.), then click "Other" (2.). In the "Open Perspective" dialogue select "LML Perspective" (3.) and click "OK" (4.).

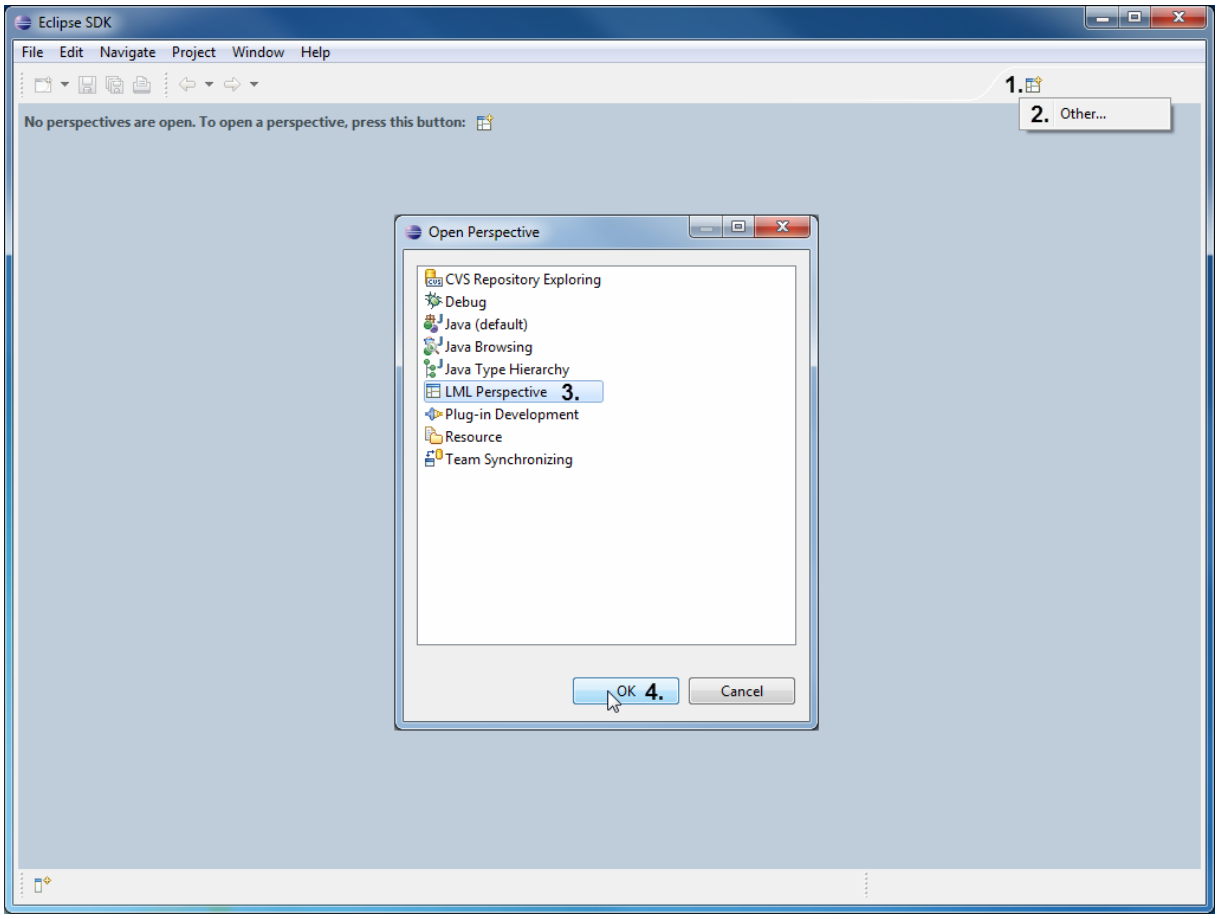

Figure 34: Opening the LML perspective.

A new project is required to which the diagram file can be added. Right-click in the "Project Explorer" (Figure 35, 1.) and select "New" → "Project..." in the context menu (2). The "New Project" wizard pops up. Extend the "General" node and select "Project" (3.). Now press "Next >" (4.). On the next page enter a project name and press "Finish".

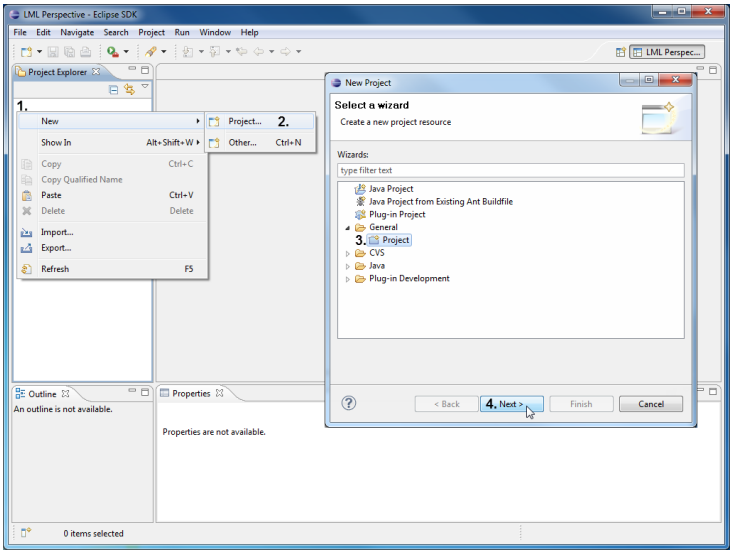

Figure 35: Creating a new empty project.

To add the diagram file to the project, right-click on it in the "Project Explorer" (Figure 36, 1.) and select "New" → "Other ..." (2.) from the context menu. The "New" wizard pops up. Enter "LML" into the text box at the top of this dialogue (3.). Select the "LML Diagram" node (4.) and click "Next >" (5.). On the next page enter the diagram file name and press "Finish". The newly created diagram is now opened in Eclipse.

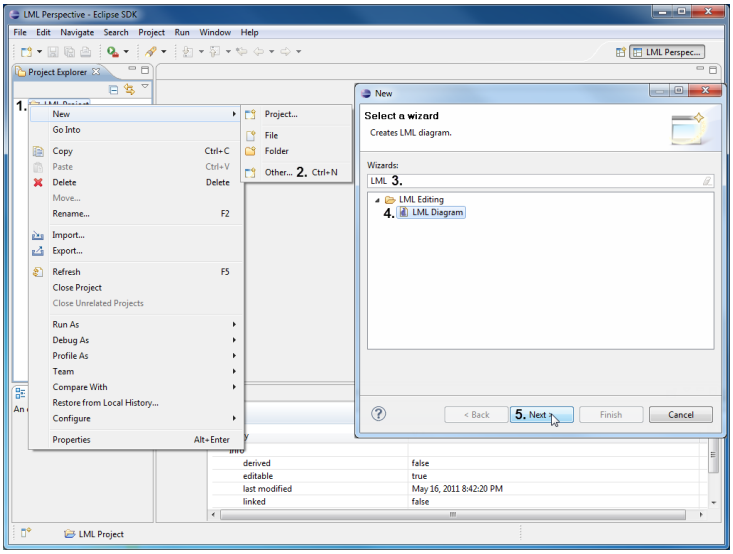

Figure 36: Creating a new model.

A.3. Walkthrough: The First Ontology

Figure 37 shows the workbench with a diagram opened. The "Palette" on the right is used to select the elements that are added to the diagram. In the editor, selected elements can be edited by using the "Properties" view at the bottom. The "Project Explorer" on the left shows the containment tree when the file is expanded. A miniature overview of the diagram is given in the "Outline" in the bottom left.

Select the "Ontology" element from the "Palette" on the right (Figure 37, 1.). Then click on the location where the element shall be placed in the editor (2.).

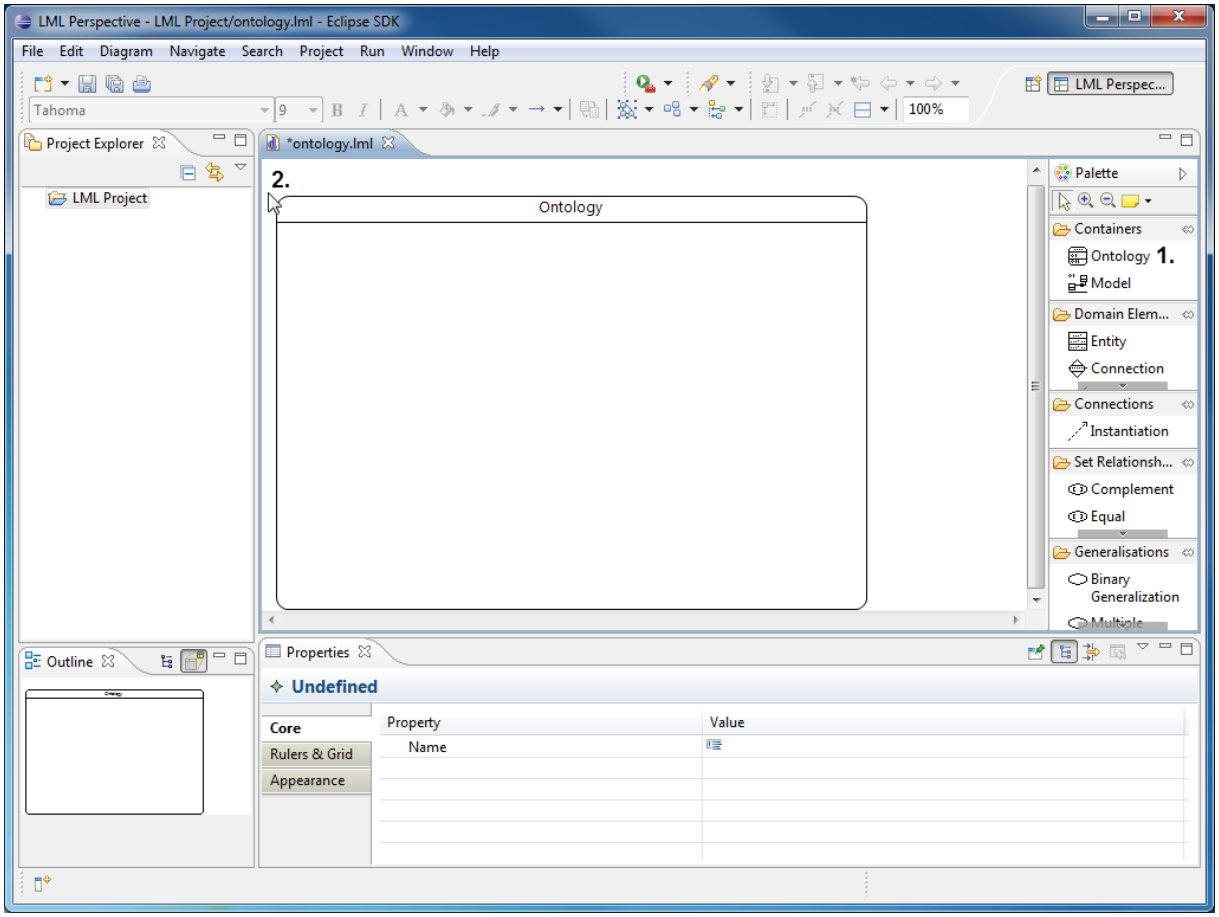

Figure 37: Adding an ontology to the diagram.

Now add the first ontological level to the ontology. Select “Model” in the “Palette” (Figure 38, 1.) and click anywhere into the ontology (2.). The diagram looks now as displayed in Figure 38.

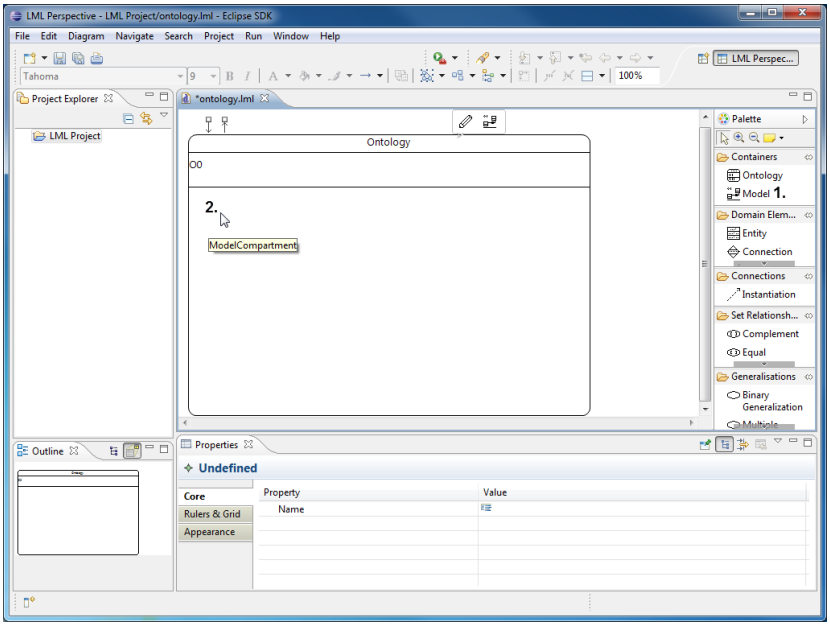

Figure 38: Adding a model to an ontology.

Start adding content to the ontological level. Select an “Entity” from the right (Figure 39, 1.), and click in the model at the position where it shall be placed (2.). Repeat this step to add a second entity.

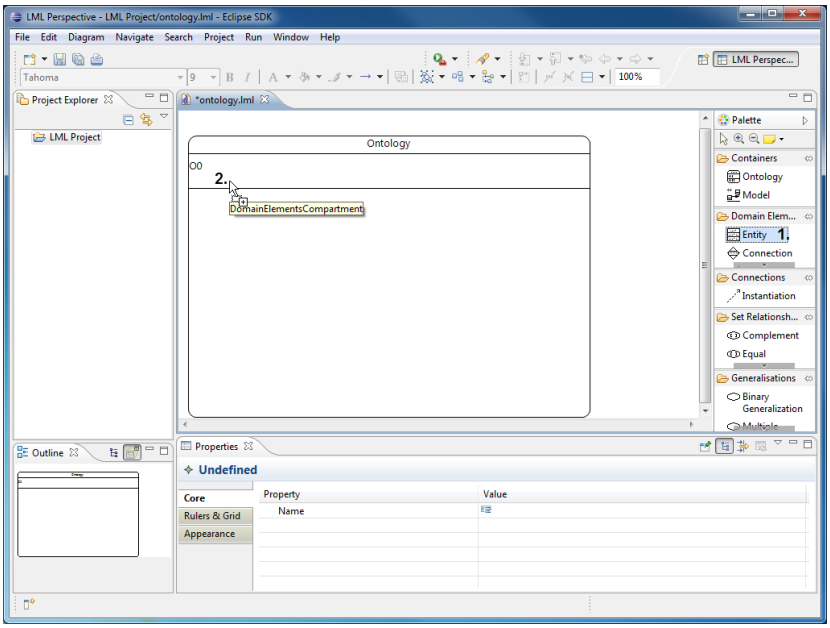

Figure 39: Adding an entity to a model.

Add a connection to the diagram which will later connect the two entities. Select the "Connection" in the "Palette" (Figure 40, 1.) and click between the two entities (2.). In the next step the entities get connected with the connection.

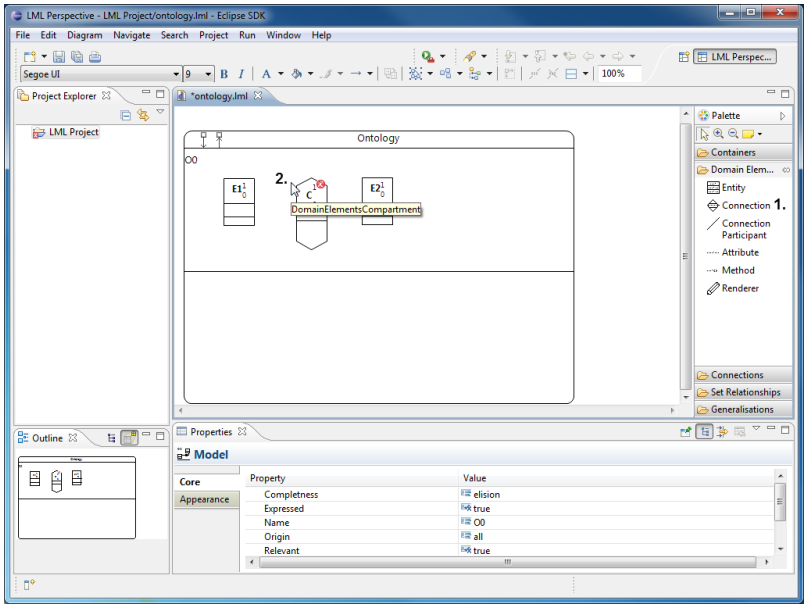

Figure 40: Adding a connection to the diagram.

Select "Connection Participant" in the "Palette". Start drawing at the connection and drag the connection line to the target entity. Then release the left mouse button. The connection is now connected to the entity through a thin black line. If two items are not allowed to be connected with each other, the mouse pointer indicates this.

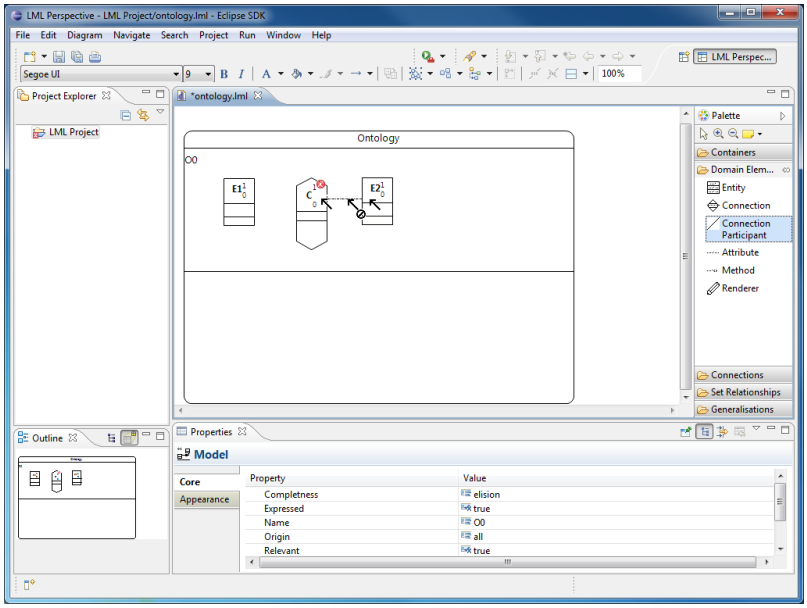

Figure 41: Connecting entities with a connection.

To toggle the connection, right-click on the connection (Figure 42, 1.) and choose "Toggle Node" (2.). The connection now appears as a small black rectangle. Repeating these steps will let the connection appear in its exploded form again.

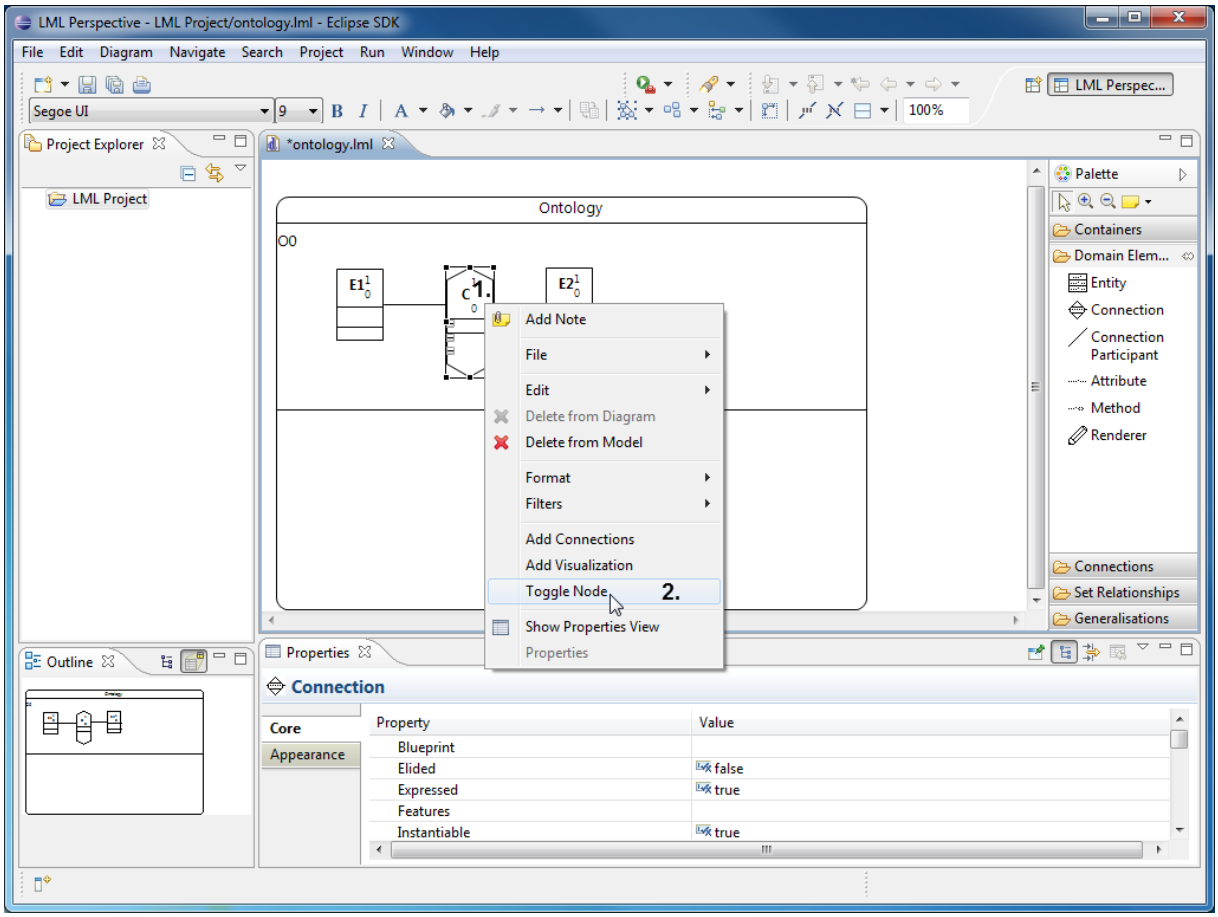

Figure 42: Toggling a connection.

Generalizations and set relationships can be used in the same way as connections. Now the basics of the editor have been explained, it should be clear that the editor is behaving like a standard CASE tool. The next chapter gives an overview of how to use visualizers to manipulate the LML's concrete syntax.

A.4. Walkthrough: Using Visualizers

To use visualizers, the visualizers must be displayed in the diagram. To display visualizers for all Elements within a model select it (Figure 43, 1.) and change the value of the "Visualizers Shown" entry from "none" to "all" (2.) in the "Properties" view. The visualizers are now visible.

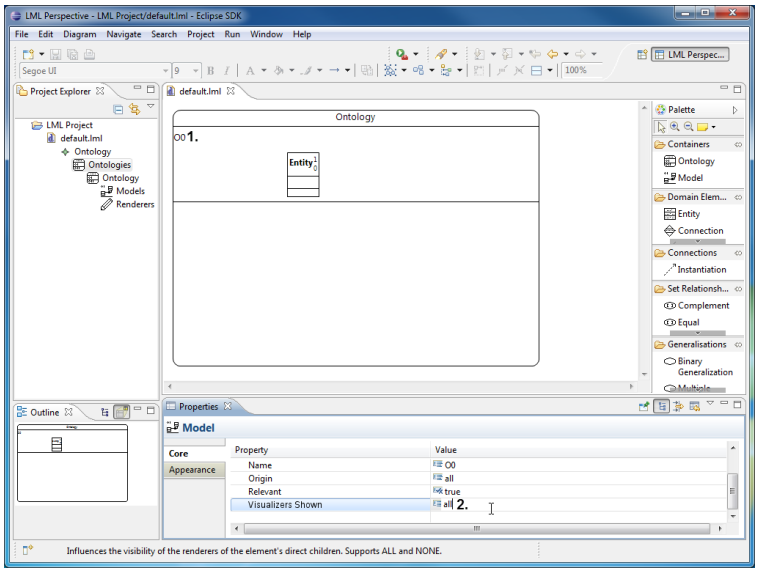

Figure 43: Showing all visualizers in a model.

Select the visualizer (Figure 44, 1.) of the element which shall be manipulated. Then select the "Attributes" row in the "Properties" view and press "..." (2.) to start editing them.

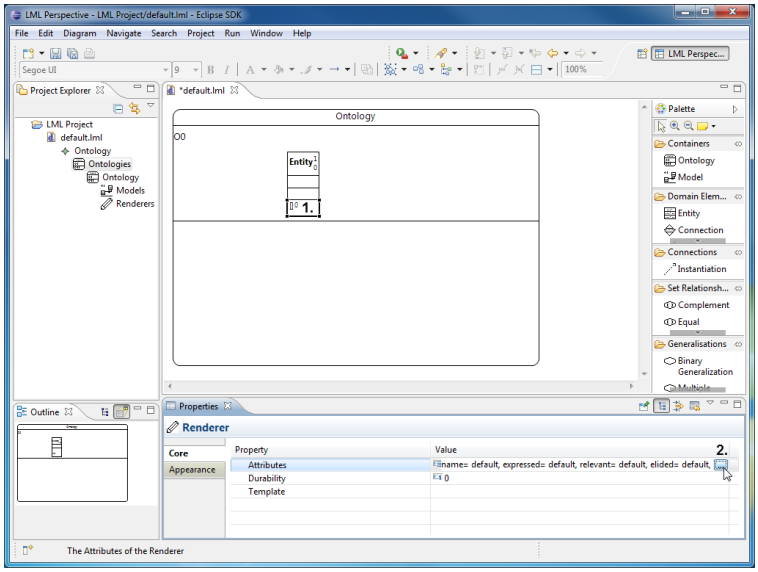

Figure 44: Selecting a visualizer.

Select the trait to be edited (Figure 45, 1.). Press “Remove” (2.) to edit the value. Now change the value from “name= default” to “name= tvs” (Figure 46, 1.) and click “Add” (2.). Then press “OK” (3.).

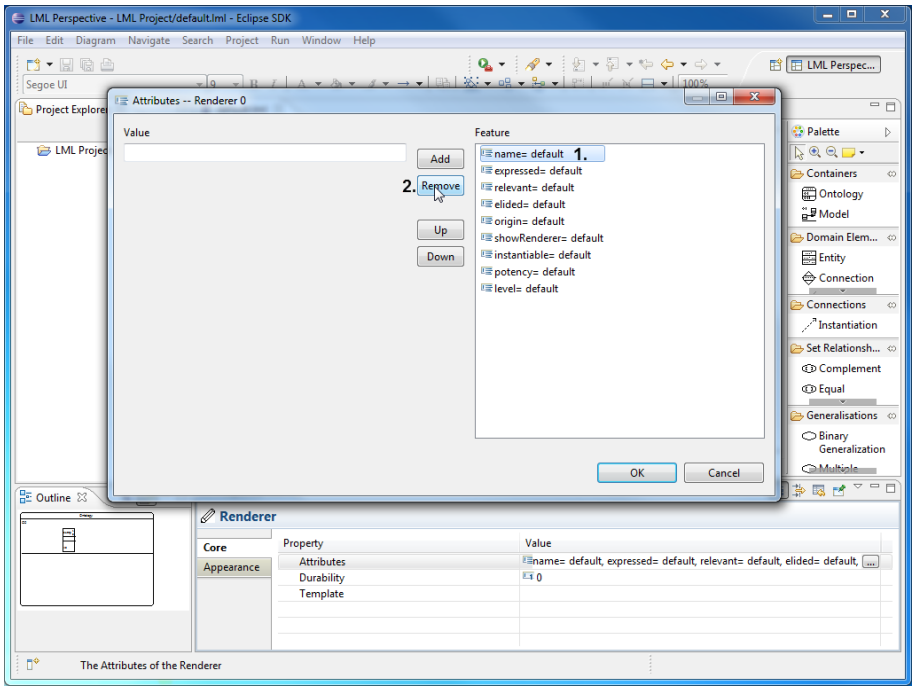

Figure 45: Editing an attribute of a visualizer (part 1).

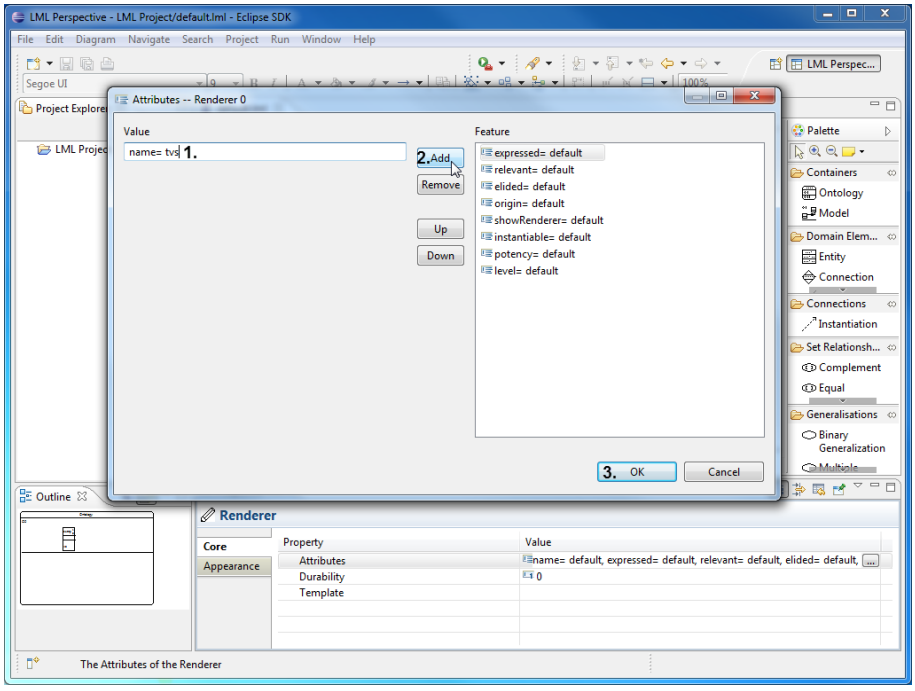

Figure 46: Editing an attribute of a visualizer (part 2).

The visualizer now manipulates the entity's concrete syntax. The name is not displayed in the header compartment anymore, but in the TVS as displayed in Figure 47. For all traits the values "default", "tvs", "noshow" and "max" are available. To hide the visualizer again select the model (Figure 48, 1.) and change the "Visualizers Shown" trait back from "all" to "none" (2.).

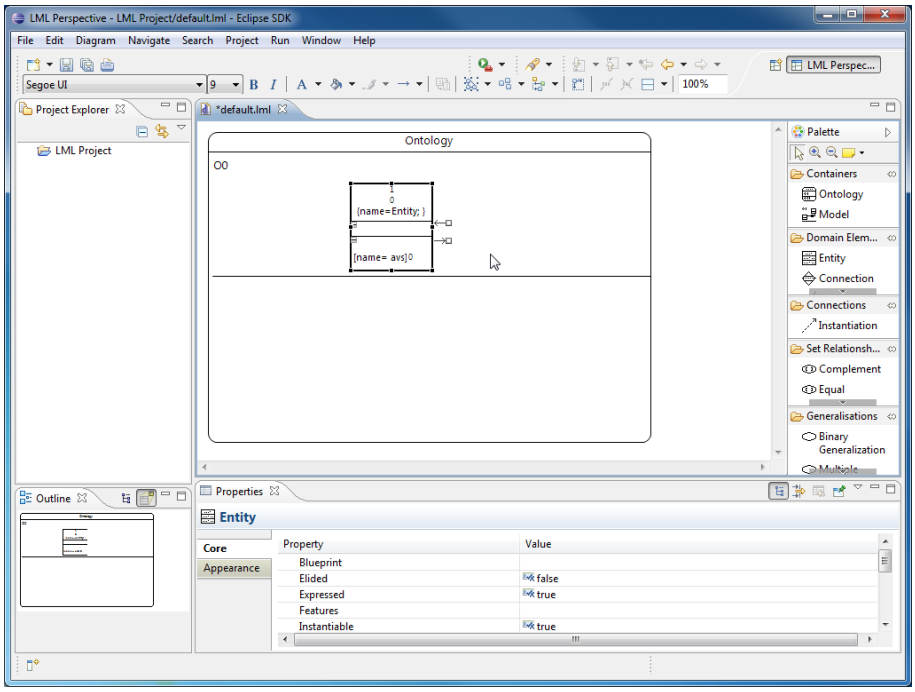

Figure 47: The entity manipulated by the visualizer.

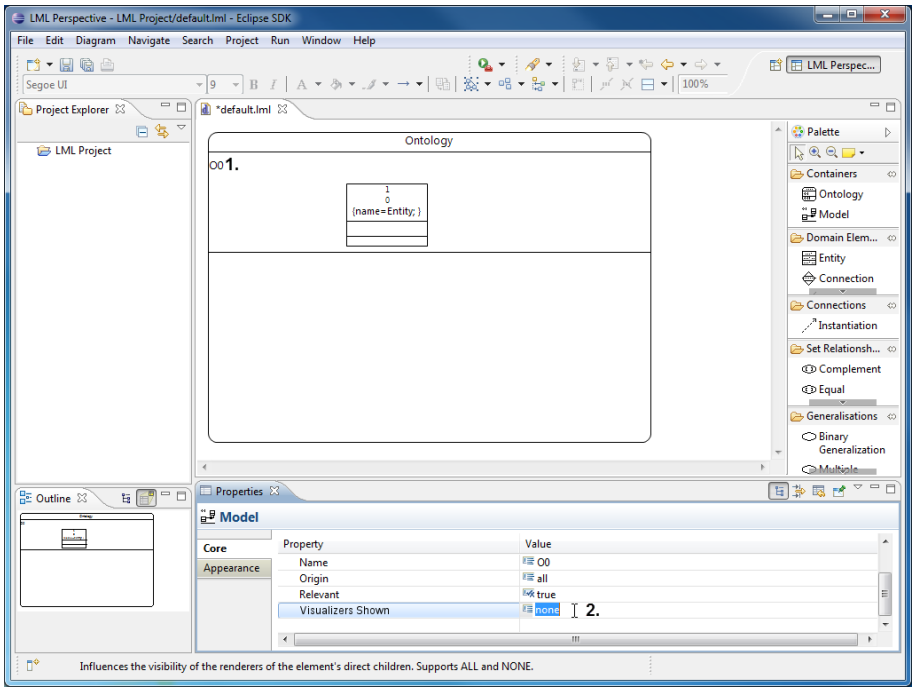

Figure 48: Hiding all visualizers in a model.

Ehrenwörtliche Erklärung

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

Mannheim, Mai 2011 Unterschrift

Abtretungserklärung

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

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

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

Mannheim, Mai 2011 Unterschrift