Deep, Seamless, Multi-format, Multi-notation Definition and Use of Domain-specific Languages

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

vorgelegt von

Diplom-Wirtschaftsinformatiker Ralph Gerbig
aus Worms

Mannheim, 2017

Dekan:	Professor Dr. Heinz Jürgen Müller, Unversität Mannheim
Referent:	Professor Dr. Colin Atkinson, Unversität Mannheim
Korreferent:	Professor Dr. Juan de Lara, Universidad Autónoma de Madrid

Tag der mündlichen Prüfung: 24. Februar 2017

To Valerie.

Acknowledgments

I thank Colin Atkinson for the supervision and the support he provided during the last six exciting years. We had a lot of open minded and fruitful discussions and he gave invaluable guidance and feedback on this dissertation and the Melanee tool.

I also thank Juan de Lara, my second supervisor, for his input to this dissertation and the joint research. Especially, I want to thank him for hosting one of my students at the Universidad Autónoma de Madrid and the support he gave to this student during his master thesis.

I thank all the students who contributed towards the success of this dissertation in the form of seminars and bachelor/master theses and as student assistants. Especially, I want to thank Nicole Wesemeyer, Katharina Markert, Daniel Gritzner and Dominik Kantner for their outstanding contributions to the Melanee tool.

I thank Bastian Kennel, for supervising my diploma (cf. master) thesis on the topic of deep modeling. This built the foundation for all following research leading to the Melanee tool as it is today and the approach described in the dissertation.

I thank Mathias Fritzsche for the opportunity to join him for an internship at SAP Research in Belfast where I had my first taste of research, model-driven development and Eclipse Plug-in development.

Finally, a special thanks goes to my beautiful wife Valerie for all the sacrifices she made and her support while I worked on this dissertation.

Abstract

Domain-specific modeling has gained increased attention in industry and academia in recent years leading to the emergence of numerous highly specialized, domain-specific modeling approaches and tools. However, the tools available today focus on one specific aspect of domain-specific modeling and tend to be isolated and difficult to integrate with each other. For instance, some tools focus exclusively on graphical, diagrammatic languages, while others focus on textual languages. The problems caused by this heterogeneous tool landscape become most evident when attempting to use multiple formats together.

A first step towards integrating different modeling formats within one single environment has been taken by the latest projective modeling environments which support the embedding of non-textual formats (e.g. tables) into textual model editors. However, in these modeling environments the textual format dominates the embedded formats. Moreover, all of these tools are based on *two-level* modeling technologies which are limited to one, hard-wired metamodel describing the modeling language and one instance level describing the user model expressed in terms of the language. In addition to the metamodel, the concrete syntax features offered by today's modeling tools are typically hardwired as well. When multiple notations are available, modelers usually have to decide for one notation before starting. To view the model in an alternative notation, the model has to be opened in a second editor.

The approach presented in this thesis for the *deep, seamless, multi-format, multi-notation definition and use of domain-specific languages* overcomes the aforementioned weaknesses. First, it treats all formats equally, no matter whether text, diagram or some other format. This allows different formats to be used side-by-side, as desired, without influencing one another negatively. Second, it allows a given format to be visualized in multiple notations in one editor, side-by-side. Third, the inherently deep architecture allows deep visualizations to be defined spanning as many classification levels as needed for language definition and use. Fourth, language definitions and visualizations are *soft*, like language applications, and can be changed at any time by modelers. The approach has been validated by means of an Eclipse EMF-based prototype implementation, called Melanee, and applied to a running example motivated by the ArchiMate Enterprise Architecture Modeling standard.

Zusammenfassung

Domänenspezifische Modellierung hat in letzter Zeit hohe Aufmerksamkeit von Industrie und Forschung bekommen. Dies führte zur Entwicklung von vielen spezialisierten domänenspezifischen Modellierungswerkzeugen und Ansätzen. Diese Werkzeuge fokussieren sich jedoch nur auf einzelne Aspekte der domänenspezifischen Modellierung. Weiter sind diese isoliert und teils schwer miteinander integrierbar, da sich beispielsweise manche Werkzeuge nur auf grafische Modellierung und andere Werkzeuge auf textuelle Modellierung fokussieren. Beim Versuch die verschiedenen Formate zu kombinieren werden die Probleme, die durch diese heterogene Werkzeuglandschaft entstehen, sichtbar.

Einen ersten Schritt zur Integration verschiedener Formate innerhalb eines Editors haben die neuesten projektionalen Modellierungsumgebungen unternommen. Diese unterstützen das Einbetten von nicht textuellen Formaten (z.B. Tabellen) in textuelle Editoren. Jedoch dominiert das textuelle Format hierbei stark. Auch basieren all diese Werkzeuge auf einer *zwei-level* Architektur. Diese Architektur ist beschränkt auf ein statisches Metamodell, welches die Modellierungssprache beschreibt, und auf ein Instanzmodell, welches das Benutzermodel mit Konzepten des Metamodells ausdrückt. Nicht nur das Metamodell, sondern auch die Funktionalitäten im Bereich der konkreten Syntax sind in den bereitgestellten Werkzeugen fest verankert. Wenn mehrere Notationen verfügbar sind muss sich der Benutzer in der Regel für eine von diesen Notationen vor dem Modellieren entscheiden. Um eine alternative Notation zu nutzen muss er das bearbeitete Modell in einem zweiten Editor öffnen.

Der in dieser Arbeit präsentierte *deep, seamless, multi-format, multi-notation definition and use of domain-specific languages* Ansatz überwindet die beschriebenen Schwächen. Dieser behandelt alle Formate gleichwertig egal ob Text, Diagramm etc. Hierbei können die verschiedenen Formate parallel benutzt werden ohne sich negativ zu beeinflussen. Auch können die einzelnen Notationen innerhalb eines Formats nebeneinander benutzt werden. Durch die durchgängig tiefe Architektur können Sprachen zusammen mit ihrer konkreten Syntax über mehrere Level hinweg definiert werden. Da Sprachdefinition und Visualisierung *soft* sind können Benutzer diese zu jeder Zeit ändern. Der Ansatz wird durch einen EMF-basierten Prototypen, Melanee, demonstriert und auf ein laufendes Beispiel welches sich an ArchiMate orientiert angewandt.

Contents

Glossary

AST	Abstract Syntax Tree
AToMPM	A Tool for Multi-Paradigm Modeling
BNF	Backus Normal Form
BPMN	Businesss Process Model and Notation
DPF	Diagram Predicate Framework
DSML	Domain-specific Modeling Language
EAM	Enterprise Architecture Model
EMF	Eclipse Modeling Framework
EMOF	Essential Meta-Object Facility
ER	Entity-Relationship
EVL	Epsilon Validation Language
GEF	Graphical Editing Framework
GME	Generic Modeling Environment
GMF	Graphical Modeling Framework
GMF-R	Graphical Modeling Framework Runtime
GMF-T	Graphical Modeling Framework Tooling
HUTN	Human-Usable Textual Notation
IT	Information Technology
LML	Level-agnostic Modeling Language
M2T	Model-to-Text
MDSD	Model-driven Software Development
Melanee	Multi-level and Ontology Engineering Environment
MVC	Model View Controller Pattern
OCA	Orthogonal Classification Architecture

OCL	Object Constraint Language
OMG	Object Management Group
OMME	Open MetaModeling Environment
OPF	OPEN Process Framework
PLM	Pan-level Model
SPEM	Process Engineering Metamodel
SVG	Scalable Vector Graphic
SWT	Standard Widget Toolkit
TVS	Trait Value Specification
UDL	User-defined Language
UML	Unified Modeling Language
VPSM	View Point Specification Model
xMOF	Executable Meta-Object Facility
YACC	Yet Another Compiler-Compiler

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In recent years Model-Driven Software Development (MDSD) has received growing attention in the domain of software engineering. In Gartner's hype cycle, published in 2006 [87], it is listed as one of the key emerging technologies alongside corporate semantic webs. Both technologies have a predicted time to mainstream adoption of 5 to 10 years. This assessment is reinforced by the steady rise in modeling standards since the first modeling languages were defined in the early 1920's, cf. [223]. Today there is a mature tool industry supporting the rapid creation of small and highly specialized domain-specific modeling languages (DSML) [60, 126]. The success of MDSD is evidenced by its central role in one of the most widely used open source development platforms — Eclipse (used by 42% of java developers [123]). In the latest release, Eclipse 4, MDSD has become such an integral part of the platform that the bulk of Eclipse applications can be represented using Eclipse modeling technology. The number of modeling projects supported by big companies like SAP, Oracle, Red Hat and IBM, as seen in [65], also demonstrates the significant interest in MDSD in the mainstream software industry.

1.1 Problem Fields of Language Engineering

Despite their success, there are some areas in which contemporary MDSD technologies still have some significant weaknesses. The following paragraphs

identify some of the problems documented in the literature and evident in today's modeling tools.

P1: Domains with Multiple Classification Levels Domains which feature more than one type and one instance level are referred to as deep domains. All current practical modeling frameworks are based on an architecture in which statically defined types are deployed to a tool and used to dynamically create user models at the instance level. This type/instance modeling dichotomy introduces accidental complexity [38] when used to model deep domains which inherently feature more than one type/instance level pair. To a certain extent workarounds can be used to circumvent this artificial type/instance level pair restriction [151]. However, as the number of domains and use cases featuring multiple classification levels grows, such workaround solutions become less acceptable.

Atkinson and Kühne highlight the existence of domains with more than one type/instance level pair when motivating the transition from the Unified Modeling Language (UML) infrastructure to modeling architectures featuring an arbitrary number of classification levels. In [31] they argue that multi-level modeling can reduce accidental complexity in the context of a product modeling language. The specific example they present focuses on modeling different product kinds, types and instances offered by a company. Other authors have also published variants of this multi-level modeling approach and have applied it to other domains featuring multiple classification levels. For example, de Lara et al. highlights the need for domain-specific metamodeling languages in [51] (i.e. metamodeling languages which are tailored for one specific purpose such as creating transformations) while Frank describes the advantages of multi-level modeling for enterprise architecture models (EAM) [82]. Other authors have used multi-level modeling approaches in the definitions of standards such as the Software *Process Engineering Metamodel* (SPEM), *OPEN Process Framework* (OPF) [92] and the *ISO 15926* standard [119].

In [151] de Lara et al. analyze more than 400 metamodels for the use of workarounds to accommodate multiple classification levels when using traditional modeling technologies. Overall they identify five common patterns

often repeated in the analyzed models. Furthermore, recent publications by the author of this thesis [14, 15] classify workarounds applied to single type/instance level pair modeling languages to improve the extensibility of deployed languages at the type level and propose a framework featuring multiple classification levels.

The importance of multi-level modeling is not only described in academic publications, but is also evidenced by the large number of multi-level modeling tools that have emerged in recent years. Two of the earliest multi-level modeling tools are MetaDepth [50], focusing on textual multi-level modeling, and Melanee [8] focusing on graphical multi-level modeling [89]. Other tools developed more recently include Modelverse [227], Diagram Predicate Framework (DPF) Workbench [148] and Open MetaModeling Environment (OMME) [237].

P2: Model Life Cycle Support Computer languages, like software, are information artifacts [79] and hence, have a similar life-cycle which includes phases such as: 1. development, 2. maintenance and 3. evolution. Like software, modeling languages are also often developed in an iterative style in which early prototypes are iteratively refined to the final product. Empirical research conducted in [109] showed that, for example the metamodel of the Graphical Modeling Framework (GMF) was changed 107 times in its first year of development. This corresponds to one change every four days. Similarly [128] identified 238 changes in the transition from UML version 1.5 to 2.0. Thus, modeling frameworks which allow the rapid prototyping of languages without the need for frequent re-compilation and re-deployment steps are most suited to the development of a new modeling language. During this process, types influencing other classification levels are steadily changed, and the affected classification levels have to co-evolve with their typing levels. Keeping track of all this co-evolution manually can be very time intensive and error prone, so a modeling framework should ideally support model evolution out-of-the-box and inherently support the iterative development of modeling languages.

Standardized modeling languages often do not fit the needs of a particular organization in an optimal way and the use cases for which they are used often change over time. An empirical study of the way languages are used

and evolved in enterprises was conducted by Linden et al. [226] in 2014. Evidence from [109] suggests that after change requests by users, technological change is the second biggest driver of model evolution. Hence support for the maintenance and evolution phases of a modeling language are essential. This support has to deal with the evolution of deployed models written in, but decoupled from, a language that is changed over time. Various papers on this topic of decoupled model co-evolutions have been published, e.g. [128, 132, 165, 199], and tools such as [39, 108] have been built to provide appropriate support.

Support for native model evolution is even more important in environments where a modeler interacts with more than one classification level at the same time such as in a multi-level modeling environment. Changes made at an arbitrary level can immediately affect large parts of a model at all other classification levels. As described in [16] even changes to small models can cause a huge amount of manual model evolution effort.

P3: Optimal Model Editing and Viewing The optimal style of interaction with a model depends on the task that has to be fulfilled by the modelers. Technical experts, who often have to enter large amounts of information, usually prefer textual representations, while business users, who require a less technical way of interacting with information, usually prefer tabular or form-based representations. On the other hand, when communicating the structure of a large, complex system diagrammatic models are often preferred. Today, most non-textual modeling tools support at least two model representation formats, the format with which a human user views and enters information (e.g. tabular, diagrammatic etc.) and the format which tools use to interchange data (e.g. XML). The need to represent models in various formats at the same time has long been recognized in industry and is supported by tools such as JetBrains MPS [41, 232, 234] or Intentional Domain Workbench [115]. These tools allow a model to be edited in a diagrammatic, textual and tabular style simultaneously. Also languages such as UML-RT [209, 210] implemented by Papyrus-RT [188, 196] which mix C++ code and graphical UML diagrams highlight this need.

P4: Multiple Stakeholder Support A given model is sometimes used by multiple groups of stakeholders. For example one group of stakeholders may want to view a model in a UML class diagram notation when doing software modeling while another group may want to view it in an Entity-Relationship (ER) [45] notation when doing data modeling. Each group of stakeholders has different requirements on the model and wants to use highly specialized notations supporting their use cases. In a tool it should therefore be possible to show a model in the particular notation desired by the stakeholder working with it. This is particularly useful when experts from different domains have to work with each other in an interdisciplinary setting. In such cases it is useful to be able to switch notations on-the-fly and view them side by side in the same tool. However, to accommodate cases when the notation of a domain expert is not defined it is advantageous to have a generally agreed fallback modeling language that can be understood by domain and non domain experts alike. Such a language should be based on the proven modeling contentions popularized by the UML and should reinforce the information encoded in the model using as much text as possible. The problem of providing support for multiple stakeholders has been recognized by industry and is mainly being tackled in MetaEdit+ [224] and by so-called view-based modeling tools, such as Eclipse Sirius [211, 231].

P5: Representation of Model States In some domains it is mandatory to customize the visualization of a model based on the state of the underlying system. An example of a domain requiring such context-sensitive modeling is the simulation and execution of models. In this domain a model needs to express such things as whether a model element has achieved a simulation target or what state the currently simulated system is in. An example of a tool that supports the representation of model states during a simulation is *A Tool for Multi-Paradigm Modeling* (AToMPM) [219] or the Executable Meta-Object Facility (xMOF) tooling [164].

P6: Abstraction-aware Visualization Often model elements are visualized based on their level of abstraction. A model element's level of abstraction depends on its position in the classification and inheritance hierarchies. Types

are more abstract than instances and superclasses are more abstract than their subclasses. This level of abstraction is represented by visualizations which become more concrete with the decreasing level of abstraction. Superclasses and meta types for example can be represented by generic, very abstract visualizations which are refined by subclasses and instances to visually indicate their level of abstraction.

P7: Complex Domain Rules Domains often feature complex rules which are reflected in the visualization of a model but not expressible through the language constructs available in graphically supported metamodeling languages. Such languages, e.g. Ecore or the Essential Meta-Object Facility (EMOF), focus on constructs from object-oriented programming like class, attribute, method and associations. Using these metamodeling frameworks, for example, it is possible to express the fact that a plane can have a limited number of passengers using association multiplicities. However, a rule that cannot be expressed in such a language but can influence the visualization of a model by highlighting model elements is: *The total weight of all passengers in a plane must be lower than the capacity of the plane in kilograms.* For this purpose, modeling tools need to be accompanied by constraint languages such as the Object Constraint Language (OCL) [184] or Epsilon Validation Language (EVL)[139]. These languages are used to express the kind of rules that are not expressible through the limited features of a metamodeling language based on the concepts of the aforementioned subset of object-oriented programming.

1.2 Requirements for Language Workbenches

Even though the problems previously outlined are known to academia and tool vendors there is no tool tackling all these problems in an integrated way at the time of writing. This section provides an overview of the requirements which need to be fulfilled by a tool that effectively and intuitively solves all the aforementioned problems in the domain of user-defined language (UDL) creation.

R1: Deep Modeling A language supporting deep modeling allows modelers to define as many classification levels as needed in a uniform way, to optimally represent the problem domain. A language workbench which supports deep modeling offers all classification levels in the same style to a modeler without the need to compile or deploy parts of a model. This allows deep domains (**P1**) to be captured and lays the foundation for natively supporting the full life cycle of a model (**P2**).

R2: Seamless Modeling Seamless modeling describes the ability to model at any classification level with all changes taking immediate effect on all other classification levels. This is usually achieved by adopting a deep modeling architecture (R1). Changes which effect more than one classification level are, however, hard for modelers to keep track of. Thus modelers need tool support for *real* seamless modeling to fully and immediately support all appropriate evolution operations needed during a model's life cycle (**P2**).

R3: Multi-format Modeling Multi-format modeling describes the editing and viewing of one model in different styles such as graph-based (diagrammatic), text-based, table-based or form-based. These styles are referred to as the representation format. All formats need to be equally supported by the modeling language workbench and need to be seamlessly integrated with each other. Editing information in one format should not negatively influence any other format. For example, editing a model in a textual format should not break the layout information in a representation of the model in a diagrammatic format. This requirement allows different stakeholders to view and edit a model in the most suitable way for the task in hand (**P3**).

R4: Multi-notation Modeling Multi-notation modeling offers the ability to edit one model using more than one notation at the same time. Notations can be switched on-the-fly and viewed side-by-side. A generally agreed fallback syntax is provided in cases where no alternate domain-specific language is defined. This allows multiple stakeholders to work individually on a model using their own domain-specific language but to communicate with experts from different domains using the general purpose language (**P4**).

R5: Context-sensitive Visualization Context-sensitive visualization controls the way a model is represented based on the context (i.e. state) of the underlying system. This context can be expressed by attributes in a model of a system. An example of context-sensitive visualization is a red background for a business process model if its *failed* attribute is set to true. To support this it is important to allow the concrete syntax used to represent model content to be determined using dynamically calculated parameters as well as statically defined values. In essence, this feature allows the state of a system instantiated from a model to be visualized at run-time (**P5**).

R6: Aspect-oriented Concrete Syntax Definition Aspect-oriented concrete syntax definition transfers the concept of aspect-oriented programming to the domain of concrete syntax definition. Parts of concrete syntax definitions can be declared as so-called join points. Model elements in the inheritance and classification hierarchies of a model element can then contribute aspects to these join points. This allows each part of a concrete syntax to be defined at the level of abstraction where it most naturally fits in a deep modeling language definition. The requirement supports abstraction-aware visualization of model elements (**P6**).

R7: Constraint Languages Supporting Deep Visualization Deep constraint languages enable a modeler to define constraints on deep models. These constraints not only exploit and complement the deep modeling approach but also support scoping across multiple classification levels. Furthermore, a deep constraint language has to introduce new functions such as checking classification and retrieving instances in a multi-level aware manner. Deep constraint languages make it possible to define complex domain rules for model visualization which are not expressible by metamodeling languages focusing on object-oriented constructs only (**P7**). Furthermore, this requirement supports the requirements **R5** and **R6** which use deep constraint languages for the calculation of context-sensitive visualization information and the expression of application conditions of aspects.

1.3 Contribution

The contribution of this work is an approach to modeling which fulfills the requirements R1-R7 to address problems P1-P7. To our knowledge no modeling technology or tool is capable of this at the time of writing. This approach is based on the deep modeling paradigm, operationally formalized for the first time in [127], the strict application of projectional editing and a visualization search algorithm which finds a suitable, abstraction-aware visualization for all model elements across all levels in a deep model.

To demonstrate the feasibility of the approach a prototype has been implemented — the deep modeling language workbench called *Melanee* [8]. This prototype supports 1. the creation of deep models, 2. assisted seamless-modeling, 3. multi-notation modeling 4. diagrammatic, textual, tabular and form-based multi-format modeling, 5. context-sensitive concrete syntax 6. aspect-oriented definition of concrete syntax, and 7. a deep OCL dialect. Melanee is the first editor to support graphical deep modeling based on the orthogonal classification architecture (OCA) [30] and is at the time of writing the only available language workbench that implements all aforementioned requirements.

1.4 Research Communication

The research conducted during this work has been presented at various international conferences and in several journals. It was first presented as a poster at Modeling Wizards 2012, and in an info booth and live demos at the 2012 MODELS Conference. Others publications have appeared at the German *Modellierung* conference's co-located workshops [13], the European Conference on Modeling Foundations and Applications (ECMFA) [10, 16] and co-located workshops [9, 21, 24], the International Conference on Software Engineering [17], the International Conference on Model Transformations [23], the Enterprise Distributed Object Computing Conference (EDOC) [14] and co-located workshops [7] and the MODELS Conference [20] and co-located workshops [8, 11, 12, 18, 19, 22]. Additionally, elements of the work have been published in the Elsevier Information Systems Journal [15] and the Springer Software and Systems Modeling Journal [25].

1.5 Outline

The rest of the thesis is structured as follows: First, the foundations of the thesis are laid out in Chapter 2. The foundations are language engineering, model-driven development and deep modeling because the contributions made here represent a delta to the state-of-practice in these fields. Furthermore, the prototype implementation presented here is based on the concepts of model-driven development. The deep modeling approach which addresses the requirement of deep domains (**R1**) is introduced in the foundations because it not only addresses **R1** but also builds the foundation for all other presented areas. A further goal of the foundations chapter is not only to introduce all concepts used throughout the work but also to create a common understanding of the technical terms used.

The foundations chapter is followed by Chapter 3 focusing on technologies which address requirements for displaying and editing models. First, user-defined visualization definition and retrieval in deep models is discussed in general. Then, multi-format visualization is discussed which addresses the requirement of multi-format modeling (**R3**) followed by a description of how the different formats are enriched with the option to model in multiple notations (**R4**). Then, an aspect-oriented approach for defining user-defined syntax across several inheritance and classification levels is presented (**R6**). Finally, the different notations are enhanced by context-sensitive functions addressing the requirement for context-sensitive visualization (**R5**).

The chapter about user-defined deep visualization of deep models is then followed by a description of the various formats supporting user-defined, deep, multi-format, multi-notation modeling. These formats are: diagram (Chapter 4), text (Chapter 5), table (Chapter 6), and form (Chapter 7).

The deep visualization approach heavily depends on deep constraints in all formats. Hence, a deep constraint language (**R7**) supporting user-defined, deep, multi-format, mutli-notation modeling is presented in Chapter 8.

Seamless modeling is then described in Chapter 9 which enables all the aforementioned technologies to be used in an efficient manner. Seamless modeling is the ability to model across multiple classification levels without the need to perform any manual or automatic deployment steps. This feature is

founded on the deep modeling approach presented in Chapter 2. The main focus of this chapter, however, is to describe the problems which result from the power of seamless modeling and point out ways of handling the complexity it causes in a deep modeling environment. It therefore addresses the requirement of seamless modeling (**R2**).

The theoretical part of the thesis's contributions is then closed with a description of the implementation of the Melanee tool in Chapter 10. This chapter describes how the tool implements the described contributions and, hence, demonstrates their feasibility. The chapter on Melanee closes with a small tutorial on how to create a deep, multi-format, multi-notation user-defined modeling language on the running ArchiMate business layer modeling example which models a company's structure.

The advantages of the approach introduced in this thesis over existing language definition workbenches are shown in a comparative evaluation (Chapter 11). In this evaluation, the company structure modeling language example covering all aforementioned requirements is created once with the Melanee tool and once with a widely-distributed, model-driven development tool stack. The advantages and disadvantages of the two solutions are compared with respect to each defined requirement for language workbenches.

The work closes with a discussion of related work in Chapter 12 followed by suggestions for future work and conclusions in Chapter 13.

Chapter 2

Foundations

This chapter describes the foundations for the work presented in the thesis, which are primarily theories from classical language engineering, model-driven language engineering and deep modeling. The first part therefore focuses on language engineering, the second part focuses on model-driven language engineering and the Object Management Group's (OMG) model-driven architecture [176], and the third part focuses on deep modeling.

2.1 Language Engineering

A software language is a "language that is created to describe and create software systems" [133]. Such languages play a key role in all branches of information technology and although the details of how they are defined and used differ, they invariably contain the same basic ingredients. In this thesis a language is defined as a set of concepts which, when combined according to a set of well defined rules and represented using a set of well defined symbols, can be used to make statements with a precise meaning about some subject (a.k.a. domain) of interest. The concepts and rules used to construct statements in a language are often referred to as the *abstract syntax* of the language, the symbols mapped to the abstract syntax and used to represent sentences are referred to as the *concrete syntax* of the language, and the meaning of statements is referred to as the *semantic domain* of the language on which

abstract syntax elements are mapped [101].

More formally speaking a language can be defined by a five-tuple $L=\{C, A, S, M_S, M_C\}$ as for example described in [44, 86]. C is the concrete syntax, A is the abstract syntax, S is the semantic domain, M_S is the semantic mapping which maps the abstract syntax to the semantic domain ($M_S: A \rightarrow S$) and M_C is the syntactic mapping which maps the concrete syntax to the abstract syntax ($M_C: C \rightarrow A$).

Since the late 1950s, abstract syntax and concrete syntax have traditionally been defined using grammar definition languages. One of the first and most widely known languages for defining textual concrete and abstract syntax is the Backus Normal Form (BNF) which was described for the first time by Gorn in [93] after appearing two years earlier for the first time in [32]. This language evolved into grammarware systems [135] such as Yet Another Compiler-Compiler (YACC) [117] which can automatically create parsers for a given grammar. These are usually accompanied by additional construction rules, called the *static semantics*, and a definition of the intended *(dynamic) semantics* of the language. Semantics can be defined by means of natural language text (informal), by constructing mathematical objects (denotational), by a precise semantic mapping of the abstract syntax concepts to another establish semantic domain (translational), by means of a reference implementation (pragmatic) or by describing how a program is interpreted as sequences through formalisms such as statecharts (operational) (cf. [133]). Today, so-called language workbenches are available which offer convenient creation of abstract syntax, concrete syntax, static semantics, dynamic semantics and corresponding tooling. Examples of such language workbenches are tools like Meta-Environment [134] from the early 1990s or the more recently published Spoofax [125]. Although the grammarware-based approach and its implementing tools focus on textual languages, tools for other language formats such as diagrammatic languages also exist today.

2.2 Model-driven Language Engineering

The goal of model-driven language engineering is to define the ingredients of a language as models. However, in order to be able to combine and vary

them in a simple and flexible way, it is not always convenient to have them bound together as in traditional grammarware approaches. Instead, it is more convenient to think of languages as being created from more or less independent components which can be mixed in more flexible ways. In this context the components of a language are referred to as the *vocabulary*, the *notation* and the *semantics*. The *vocabulary* is the set of abstract concepts from which statements in a language can be constructed and the rules by which they can be combined. This corresponds roughly to the abstract syntax and static semantics in traditional grammar-based definitions of languages. It is usually described through metamodels expressed in metamodeling languages using object-oriented concepts such as *class*, *attribute* and *reference*. Where needed these metamodels are enriched with a constraint language expressing domain rules which cannot be expressed in the used metamodeling language. The *notation* is the set of symbols used to represent concepts appearing in a statement in a language, along with the allowed mappings of those symbols to the concepts. This corresponds roughly to the concrete syntax of grammar-based language definitions, but does not contain some rules that would traditionally be defined in the grammar. The concrete syntax of a language is modeled by small, highly specialized, domain-specific modeling languages which focus on modeling concrete syntax in a certain format (e.g. diagrammatic or textual). The *semantics* contains the description of the meaning of the concepts in valid statements of the language (i.e. the mapping of meanings to concepts in the vocabulary). Below, tools which support the definition of the vocabulary, notation and semantics of a modeling language are referred to as (modeling) language workbenches.

It is also convenient to introduce the notion of *format* to represent the basic style used to visualize statements in a language. Four basic styles are commonly used in computer science — text, diagram, tree and table visualizations — but others are possible. The format in which a statement in a language is represented obviously depends on the notation used to visualize it. A notation can only support one format, but a format can be supported by many notations. For example, it is possible to have different sets of symbols to support different diagrammatic visualizations of a statement in a language, or different sets of strings to support different textual visualizations. The formats which

are supported by model-driven language engineering tools range from textual (e.g. XText [70] and EMFText [103]) and form-based (e.g. EMF Forms [72] and EMF Parsley [191]) to diagrammatic (e.g. MetaEdit [224], Sirius [231] and GMF [94]).

To allow more freedom for users to mix and match these ingredients it is convenient to use the term *language* to refer to just the first of them — the vocabulary, and to regard the others as being associated with the vocabulary. Note that this is a subtle, but fundamental, departure from grammar-based definitions of languages, where all the ingredients are regarded as characterizing parts of the language. This new interpretation of the term language makes it possible to talk about a language having more than one notation (e.g. normal and interchange syntax [133]) and more than one semantics (e.g. STATEMATE [100], fixpoint [194] and UML [181] semantics for statecharts [75]). Strictly speaking this is not possible with the grammarware approach to languages, but corresponds to every day usage of terminology. For example, it is common to refer to different textual or graphical concrete syntax of the UML or the previously mentioned semantic variants for statecharts. In both cases, what characterizes the language that has multiple concrete syntax or semantics associated with it, is the vocabulary.

An important distinction to make when discussing language engineering is the difference between predefined language components and user-defined language components. The former come predefined, *out-of-the-box* as part of the environment or tool that is used to create models, while the latter are created by users of the environment or tool for their own specific purpose. This distinction is often alluded to using terms such as *general-purpose language* and *domain-specific language*, since predefined languages are usually general-purpose and user-defined languages are usually domain-specific. However, since this does not have to be the case and the terms *general-purpose language* and *domain-specific language* are misleading, they are generally avoided in favor of *predefined language* and *user-defined language*. It is important to note that the notions of being predefined or user-defined apply to the individual components of languages as identified above (i.e. vocabulary, notation and semantics) separately. Thus, it is possible to define a new semantics and/or notation for a predefined vocabulary. A modeling tool or environment should

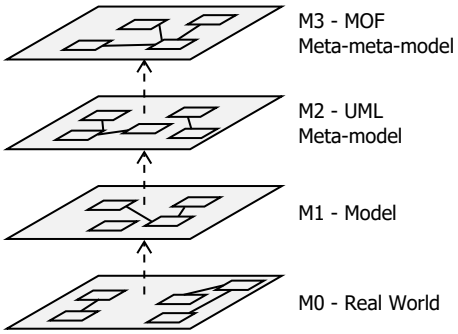


Figure 2.1: The OMG four-level architecture after [106].

offer at least one predefined language (i.e. vocabulary) with at least one predefined notation and at least one predefined semantics to be useful *out-of-the-box*.

Another distinction to make when discussing model-driven approaches to language engineering is the difference between defining a language and its associated components (*language definition*) and using a language and its associated components (*language use*). Since both involve modeling, it is easy to confuse them. However, it is important to be clear when one is referring to the use of a language (or one of its components) or the definition of a language (or one of its components) since these two activities take place at different classification levels, are supported by different tools and are executed by different actors. Languages are defined by language engineers, supported by language workbenches, and used by language users, supported by modeling workbenches. By definition, the model resulting from the use of a language resides on the levels below the definition of that language.

At the time of writing the most widely applied architecture to support model-driven language engineering is based on the four-level, OMG infrastructure shown in Figure 2.1. This infrastructure defines a stack of four model levels M_3, M_2, M_1 and M_0. Each level is described by the previous level in this stack. M_3 is an exception because it is described by itself to prevent an endless recursion. Apart from M_0 each level is effectively a metamodel for the level below. However, the levels in the OMG infrastructure are traditionally named relative to M_1. Thus, M_2 is referred to as the metamodel, M_3 as the meta-metamodel, M_1 as the model. M_0 is usually referred to as the real world.

When applying the previously defined terms to modeling language workbenches based on the OMG four-level modeling infrastructure, the predefined language is located at the highest meta-level — M_3. This predefined language is then used to create a user-defined language at level M_2 which defines its own vocabulary, notation and semantics. This language is then deployed to a tool as a predefined language where it can be used to create model content (M_1).

2.3 Deep Modeling

Deep modeling, also referred to as deep metamodeling, is an evolution of model-driven development known from the previously described OMG four-layer architecture. The fundamental difference is that deep modeling supports modeling across as many classification levels as needed to represent the domain at hand — so-called multi-level modeling. Different flavors of multi-level modeling are available depending on the primary format they use to represent deep models, such as textual (e.g. MetaDepth [50], DeepJava [146]) or diagrammatic (e.g. OMME [236]) and the architecture on which they are based. At the time of writing, the most widely adopted architecture for deep modeling is the so-called Orthogonal Classification Architecture (OCA) [30] which is usually accompanied by the notions of deep characterization [105, 147] and strict metamodeling [6].

Figure 2.2 shows the orthogonal classification architecture populated with an excerpt of a language for modeling enterprise architectures. The modeling language is based on ArchiMate [221] modeling constructs for modeling the active structure of a company. The language definition consists of the concrete incarnations of the BusinessActor extensions suggested by the ArchiMate standard [221]. These are CompanyType (Organization), Department (Organization Unit) and the EmployeeType (Individual) which is not shown here. CompanyTypes consist of DepartmentTypes and DepartmentTypes of EmployeeTypes. This language is referred to as the *company structure modeling language* in the rest of this thesis.

In contrast to the earlier presented four-layer OMG modeling infrastructure it can be observed that classification levels are present in two orthogonal dimensions. The vertically stacked linguistic classification levels, labeled L_2 -

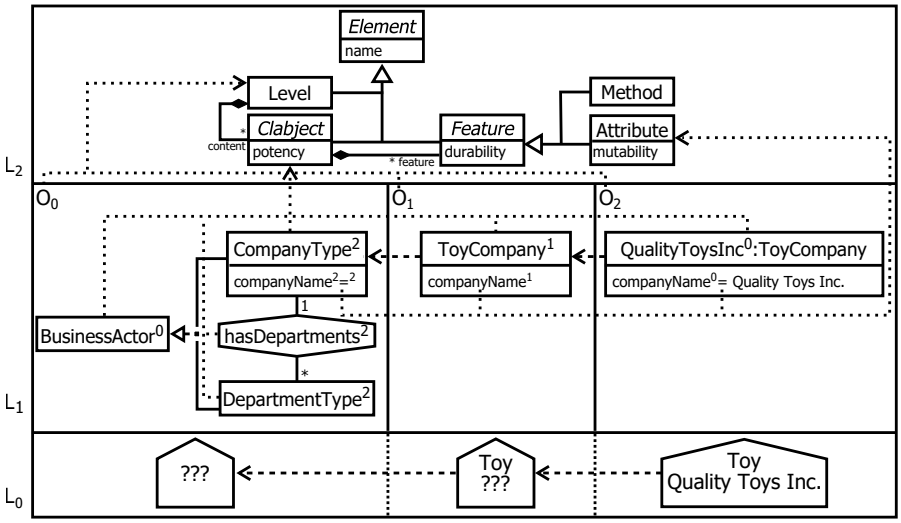


Figure 2.2: An illustration of the Orthogonal Classification Architecture.

L_0 and the horizontally stacked ontological classification levels named O_0 - O_2. This orthogonal arrangement of classification levels gives the OCA its name.

The top-most linguistic classification level, named L_2, spans the ontological classification levels, named O_0 to O_n, which collectively constitute the deep model. The L_2 level, also referred to as the pan-level model (PLM), is hard-coded into deep modeling tools and defines the predefined modeling language used to create deep model content. This predefined language ships with a predefined notation which is called the level-agnostic modeling language (LML) [27]. The LML is designed to be compatible with as many concepts as possible from existing diagrammatic notations such as the UML or ER. For example, the box-notation for model entities containing a textual representation of their attributes, as used for CompanyType in Figure 2.2, clearly originates from the UML notation whereas the hexagonal notation for connections, as used for the hasDepartments connection between CompanyType and DepartmentType in Figure 2.2, is inspired by the diamond representation of connections in the ER notation.

The ontological domain content modeled using the predefined deep modeling language is contained in the second linguistic level, L_1. This level is divided into three ontological levels O_0 to O_2. In this example three levels are shown

for illustrative reasons, but in general there can be an unlimited number of ontological classification levels. All model elements residing in the ontological levels of Figure 2.2 are linguistically classified by elements in the PLM as indicated by the vertical dotted classification arrows. Ontological classification is denoted by horizontal dashed classification arrows. Alternatively, ontological classification can be shown by placing a colon after a clabject's name followed by the name of the clabject's ontological type. An example of such a type indication is shown for QualityToysInc in Figure 2.2. The ontological entities and connections are linguistically classified as Clabjects, which is a name derived by concatenating the words class and objects to emphasize the type and instance duality of model elements in a deep model [5]. Although the model elements at the highest ontological level, O_0 and the lowest ontological level, O_2 are either types only or instances only, they are also classified as clabjects for two reasons. First, it allows all model elements to be referred to using uniform terminology, and second it allows levels to be attached to both ends of a deep model at any time.

All model elements in Figure 2.2, Clabjects and Attributes, have numbers attached as superscript next to their name. In the case of Clabjects, this number is called potency [29] and is always a non-negative Integer value (potentially including infinity). The potency of a Clabject specifies how many other ontological levels are influenced by it. In Figure 2.2, CompanyType has a potency of *two* expressing the fact that it can influence the following *two* classification levels. In other words this means that instances of CompanyType can exist at level O_1 and O_2. Here these instances are ToyCompany at level O_1 and QualityToysInc at level O_2. The potency is decreased by *one* at each subsequent instantiation step resulting in a potency of *one* for ToyCompany and *zero* for QualityToysInc. An exception to the rule of decreasing the potency by *one* at each instance occurs if the potency is a *star* value ($*$) which corresponds to infinity (or unlimited). Clabjects with $*$ potency can influence a deep model over an unlimited number of subsequent classification levels. Instances of such Clabjects can either have *star* potency or any non-negative numeric potency.

A special case for potency is illustrated by BusinessActor which has a potency of *zero* but subclasses with a potency of *two*. Such superclasses with a potency of *zero* but subclasses with potency higher than *zero* correspond to the

concept of abstract classes in the UML. Moreover, a dot can be observed at the location where the inheritance relationships between BusinessActor and its subclasses join. This shows the inheritance relationship in its collapsed, visually insignificant form. Alternatively, inheritance relationships can be shown in an expanded form as a rectangle with a curved top and bottom. This visualization allows additional information to be displayed such as an inheritance name or statements about instances, e.g. disjointness or completeness. This feature called *dotability* is also extended to connections in order to save space in cases where the connection itself does not convey additional useful information.

There are two additional forms of potency which help to support deep characterization — attribute potency (durability) and value potency (mutability). The durability, represented next to an Attribute's name, specifies over how many instantiation steps of the containing Clabject that Attribute *endures*. The mutability, represented next to the value of an Attribute, specifies over how many levels the value of an Attribute can be changed. For both, mutability and durability, the same basic rules as for Clabject potency apply. However, in the case of mutability there is the additional rule that the mutability of an Attribute must not be higher than its durability. In some cases the values for durability and mutability can be elided in a deep model. Durability can be hidden when it is the same as the potency of its Clabject and mutability is usually hidden when it is the same value as the Attribute's durability. The reduction rule for Clabject potency is also applied to durability and mutability. Additionally, in the case of mutability the value is set to *zero* if it is *zero* at the type level.

The real world represented by the deep model content is located in the lowest linguistic level, L_0. In the case of Figure 2.2 a house pictogram is used to represent the concept of a company at various levels of abstraction. At the most abstract level, O_0 the concept CompanyType in the real world is represented by a house symbol containing question marks to show that it leaves a lot unspecified. At the next level, the company concept is refined to a ToyCompany which is represented by a more detailed house symbol, a house with Toy placed in it. At the lowest level of abstraction, the actual company Quality Toys Inc. is represented by a house symbol showing the type (Toy) and the name (QualityToysInc) of the company. In the case of a real company a picture or logo of the actual company could have been used instead.

From a tool point of view two of the three linguistic levels are available to the user. The L_2 level is available to the user in the form of a palette in diagrammatic modeling environments, code completion in textual modeling environments or any other form depending on the modeling format at hand. L_1 is the level with which a user will interact most of the time as it contains the actual deep model content and thus usually occupies the largest portion of the screen. Level L_0 is not shown in a deep modeling tool because it represents the actual concepts being modeled, and thus is not a part of the model per se.

A full formalization of this deep modeling approach forming the foundation for this thesis is available in [127] which defines the linguistic metamodel, classification semantics and model checking services associated with deep models. The work described in this thesis builds upon the first prototype implementation [89] of a diagrammatic deep modeling tool based on this formalism.

In practice, the deep modeling approach blurs the borders between language definition and language use as previously defined. A user of a language defined at a more abstract classification level is not only using this language but also defines a new language for the following classification levels. The only levels where the distinction between language use and language definition can be clearly made are the most abstract ontological classification level (language definition) and the most concrete ontological classification level (language use). The blurring of the conceptual border between language definition and use also blurs the practical distinction between the tools used for both activities. In deep modeling the same tool is used by modeling language engineers and modeling languages users. Hence, the roles of modeling language user and language engineer are no longer bound to a certain classification level or tool but to the usage scenario for which the model content is developed. An actor creating a model for use as a modeling language by other actors is playing the role of a language engineer regardless of the classification level at which the modeling task is performed while an actor creating a model not intended to be used as a modeling language by other actors is playing the role of a language user, even though this task could take place at any ontological classification level.

Chapter 3

User-defined Visualization of Deep Models

The approach developed in this thesis for supporting the user-defined visualization of deep models contains six key elements. The first is deep visualization supported by a search algorithm which finds visualizations for model content to be visualized using the inheritance hierarchy, ontological classification hierarchy and linguistic classification hierarchy. The second is the capability to edit and view deep models in multiple formats, including diagram, text, form and table formats. The third is the capability to mix and match multiple notations within one format in a way that best suits the stakeholder and task at hand. The term notation refers to a specific visualization of a model element in one format. A process step, for instance, could be visualized using the activity diagram notation [61], Business Process Model and Notation (BPMN) [174] notation or the built-in predefined LML notation. The fourth is support for abstraction-aware notation definition through aspect orientation. The fifth is the ability to represent states of executed models at the instance level through context-sensitive visualizations. And sixth, the ability to use all of these features side-by-side, on-the-fly with immediate impact on the visualization of the currently viewed model.

This section introduces each of these ingredients and explains how they contribute to the overall deep visualization mechanism developed as part of

this thesis to meet the requirements outlined in Chapter 1. The following chapters then provide more details on the specific modeling formats.

3.1 Language Visualization for Deep Models

The foundations chapter made a clear distinction between language definition and language use. Language definition was described as the definition of a new language which, today, usually takes place at the meta-level. On the other hand language use is defined as the usage of a user-defined language to create a model which takes place one level below the meta-level. Usually, a language is defined using a predefined visualization and is used using a user-defined visualization. This definition and alignment of classification levels to language definition/use originates from the nature of state-of-the-art meta-modeling tools which only offer two classification levels for modeling. The one is the meta-level which is available at language definition time and then deployed into the modeling workbench. The other is the model level which instantiates the metamodel in a tool to which it has been deployed. This technical limitation which strictly separates language definition and language use leads to the traditional definitions presented previously.

Deep modeling, in contrast, blurs the border between language use and language definition. In a deep model all levels are equally available for modeling, and any model content defined at one level is immediately available for use (via instantiation) at the next level. The only ontological classification level where this is not the case is the most abstract level which is not classified by any higher level. Hence, except when modeling on the most abstract classification level, a modeler is essentially using and defining a language at the same time. Applying this principle, all the classification levels above a given level contribute to the classification of model elements and the definition of the language. Furthermore, predefined and user-defined visualizations for modeling languages have to be available at all classification levels, as desired by a modeler, and each level must be able to contribute to the visualization of model elements.

Furthermore, unlike tools based on two modeling levels (class and instance level) deep modeling tools offer enough levels to display the execution of state-

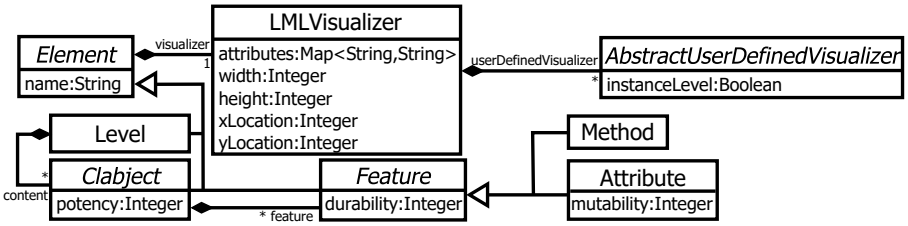


Figure 3.1: The LMLVisualizer and AbstractUserDefinedVisualizer.

ments made in a language. This information usually occupies the most concrete level of a deep model. This level, however, is not necessarily created using a modeling tool but rather through a tool interpreting the model at the type level. We refer to such a scenario as model usage too because in most cases the tool creating the instance level is controlled through a tool user and thus at the same time a model user.

This gives rise to four key requirements for deep visualization: 1. A visualization must be definable at each level, 2. a visualization definition must be available more than one level below its definition, 3. visualizations must be refinable across classification levels, and 4. a modeler must be able to choose between the predefined visualization for defining new language concepts and user-defined visualization for using a language.

3.2 Deep Visualization

Deep visualization refers to the ability to define user-defined visualizations for model content which can be applied over an unlimited number of classification levels. Furthermore, all changes to the visualization definition have immediate effect to all classification levels without the need for deployment or compilation steps. To define visualizations so-called predefined and user-defined visualizers are attached to model elements. Figure 3.1 shows an excerpt of the PLM with an enhancement allowing the configuration of the predefined visualization and the attachment of user-defined visualizations. In this version of the linguistic metamodel each linguistic Element has a predefined LMLVisualizer attached. This visualizer drives the predefined LML visualization of deep-models. The attributes map contains one key/value pair for each trait of the model element

the LMLVisualizer is attached to. The key identifies the trait to be visualized and the value represents the configuration for this trait. The configuration values are: 1. default —the default visualization of the trait, 2. noshow — the trait is not shown (i.e. hidden), 3. show — no elision rules are applied for the trait (e.g. hide durability if equal to clabject potency) and 4. tvs — show the trait in the so-called *trait value specification* underneath the clabject's designator. In the case of Clabject, for instance, the attributes map contains a key/value pair for name and potency. Detailed specifications of the predefined diagrammatic LML notation used for visualization of deep models can be found in [27, 89]. Besides the trait visualization, the location (xLocation, yLocation) and size (width, height) of the attached model element is stored in the predefined LMLVisualizer.

It is possible to replace the predefined visualization with a user-defined visualization. AbstractUserDefinedVisualizers specifying such a user-defined visualization can be attached to the LMLVisualizer for this purpose. When visualizing a model element at a specific ontological level an algorithm is applied to search for the visualization of the model element in question. This algorithm is called the visualizer search algorithm.

Espinazo-Pagán et al. [76] introduced the first visualization search algorithm for searching user-defined visualizations in the domain of *two-level* user-defined modeling languages. This algorithm was proposed in the context of textual user-defined languages for metamodels (e.g. Ecore, MOF). When visualizing a metamodel instance, the algorithm searches the classification hierarchy and the inheritance hierarchy of the instance's types for visualization descriptions. The algorithm was further improved to support deep models and the orthogonal classification hierarchy by Atkinson et al. in [26] with a focus on diagrammatic languages.

This deep form of the visualization search algorithm starts searching for a visualization at the clabject to be visualized, then searches its supertypes. If no visualization is found, the ontological types of the clabject to be visualized and their inheritance hierarchy are searched. This algorithm is applied recursively over all classification levels until the most abstract ontological classification level is reached. If no user-defined visualization is found at this level, the predefined LML notation is used as a backup for visualization. It is also

possible to mix domain-specific and general purpose renderings in one view on a model, as described for the first time by Atkinson et al. in [17]. The deep visualization search algorithm was first implemented by Gerbig in [8, 89] as part of the work presented here.

```
   Data: elementToVisualize;
   Result: A visualizer suitable for visualizing elementToVisualize
 1 types ← elementToVisualize;
 2 classification: while (type ← types.poll()) ≠ null do
 3     types ← types ∪ type.getDirectTypes();
 4     superTypes ← type ∪ type.getDirectSupertypes();
 5     inheritance: while (clabject ← superTypes.poll()) ≠ null do
 6         for udlVisualizer ∈ getUDLVisualizers(clabject) do
 7             if applicable(elementToVisualize, udlVisualizer) then
 8                 return udlVisualizer;
 9             superTypes ← superTypes ∪ clabject.getDirectSupertypes();
10             types ← types ∪ clabject.getDirectTypes();
11         end
12     end
13 end
```

Algorithm 3.1: The basic version of the visualizer search algorithm.

Algorithm 3.1 shows the basic deep visualization search algorithm. It expects the elementToVisualize clabject as input which is either an Entity or Connection to be visualized. First, the elementToVisualize is added to the types queue in line 1. Then all types are iterated using the loop labeled classification in line 2. The types of the current type are added to the types queue (line 3). Afterwards, the supertypes of the type and the current type itself are added to the superTypes queue (line 4). The superTypes queue is then iterated over (line 5) and searched for a visualizer.

All AbstractUserDefinedVisualizers of the current clabject in the inheritance tree are iterated (line 6) and checked if applicable with the applicable() function. If an AbstractUserDefinedVisualizer is applicable it is returned for visualization, otherwise the types and supertypes of the current model element are stored in the superTypes and types queues and the search is continued by further searching

the inheritance hierarchies and ontological types in the following iterations. If no result is found after searching the whole ontological classification hierarchy and inheritance hierarchy, the predefined LMLVisualizer is returned for visualization by the search algorithm.

The visualization search algorithm presented here uses queues instead of lists to store the clabject hierarchies to be searched. This ensures that a breadth first search algorithm is applied and, hence, that the nearest AbstractUserDefinedVisualizer in the classification and inheritance hierarchies is applied. If two visualizers are the same distance from the model element to be visualized the current version of the algorithm simply takes the first one discovered. It would be possible to attach concepts such as visualizer priorities to visualizers to resolve such conflicts, but the visualizer search algorithm could still find two visualizers with the same priority and distance to the clabject to be visualized. The current version therefore aims to keep the user-defined visualizer definitions simple so that language engineers should find it easy to define languages in such a way that only one visualization at a time is found by the visualizer search algorithm.

Data: elementToVisualize; udlVisualizer;
Result: Check if a visualizer is applicable
1 **if** *udlVislualizer.isInstanceLevel()* **then**
2 | **if** *isSameLevel(elementToVisualize, udlVisualizer)* **then** return false;
3 **end**
4 return true;

Algorithm 3.2: The basic version of the visualizer search algorithm's applicable function.

The basic version of the applicable function applied to user-defined visualizers is shown in Algorithm 3.2. The function expects the elementToVisualize (either an Entity or Connection) and the candidate user-defined visualizer (udlVisualizer) as input. It then checks whether the latter is applicable to the former. In the current version of the function *applicable* means to check whether the instanceLevel attribute is set to *true* and the elementToVisualize is on the same level as the udlVisualizer. If this is the case, the applicable function returns *false* because the instanceLevel attribute makes a visualizer applicable to instances

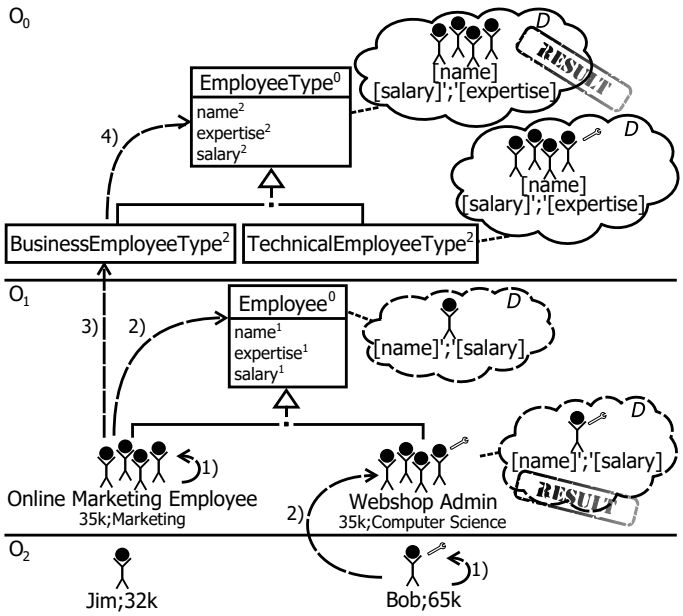


Figure 3.2: A run of the visualization search algorithm for Bob and Online Marketing Employee.

only. In all other cases the applicable function returns *true*.

A run of the basic visualizer search algorithm shown in Algorithm 3.1 is demonstrated on the company structure modeling language example in Figure 3.2. For demonstration purpose a notation close to the ArchiMate *icon* notation is chosen even though the *box* notation, visualizing model elements as boxes with the icon located in the box' upper right, is more popular. The icon notation, however, is visually more distinctive. The figure shows the trace of a request to visualize Online Marketing Employee and one to visualize Bob. The dashed arrows with a number attached indicate the order in which nodes are visited by the search algorithm. User-defined visualizers are attached to clabjects using a cloud symbol. The visualizers are defined using instances of a format-specific visualizer metamodel, here the diagrammatic format indicated by the D in the upper right of each cloud. However, to simplify the figure the intended visualization is shown graphically using the intended user-defined visualization in Figure 3.2 instead of using object specifications based on the

diagrammatic user-defined visualization definition metamodel. Text in brackets represents a mapping of an attribute's value to a label while text enclosed in single quotation marks represents a label containing static, unchangeable text. Dashed clouds indicate that the instanceLevel attribute of the visualizer is set to *true* and is thus not applied to clabjects residing at the same ontological level as the visualizer. When these rules for user-defined visualizer definitions are applied to Employee, instances of Employees are visualized as stickmen with the value of the name attribute followed by a colon and the value of the salary attribute displayed underneath the stickman icon. The visualization, however, is only applied to instances of Employee because the dashed border of the cloud representing the visualizer indicates that the instanceLevel attribute is set to *true*.

In the case of Bob, the visualizer search algorithm first looks at Bob itself for a suitable visualization. Bob, however, has no user-defined visualizer attached. Next the visualization search algorithm looks at the supertypes of Bob for a suitable visualizer. As Bob does not participate in any inheritance relationships and, thus, does not have any supertypes this step is skipped by the visualization search algorithm. The search at the ontological types follows the search of the supertypes. Bob has one ontological type, Webshop Admin, which can be derived from its visualization. This ontological type has a visualizer attached which visualizes its instances as a stickman with a wrench in the upper right-hand corner. Below the stickman, the name attribute value followed by a semicolon and the salary attribute value of the Webshop Admin are displayed. The visualizer search algorithm terminates after two steps once this user-defined visualizer for the visualization of Bob has been applied.

The second visualizer search algorithm trace for Online Marketing Employee, as displayed in Figure 3.2, involves more steps than the search for the user-defined visualization of Bob. First the visualizer search algorithm searches Online Marketing Employee itself for a visualizer but does not find one attached. The supertypes (here the abstract Employee clabject) are then searched for visualizers. Employee does have a visualizer attached which has the instanceLevel attribute set to *true* as indicated by the dashed border of the cloud representing the visualizer. Thus, because Online Marketing Employee and Employee reside at the same level, the visualizer search algorithm has to continue and

search further for a visualization of Online Marketing Employee from the ontological types of Online Marketing Employee because no supertypes are left to search. The ontological type of Online Marketing Employee, is BusinessEmployeeType which does not have a visualizer attached. Hence, the visualization search algorithm continues to search the inheritance hierarchy of BusinessEmployeeType. Its supertype, EmployeeType, has a user-defined visualizer attached. This visualizer visualizes instances of EmployeeType as a group of stickmen with the value of the name, salary and expertise attributes rendered at the bottom. This visualizer is then returned for visualization of Online Marketing Employee by the visualizer search algorithm.

3.3 Multi-format Modeling

The format of a modeling language determines whether it is visualized in a diagrammatic, textual or tabular form etc. A multi-format editor supports modeling using different formats at the same time in a seamless manner. This means that editing in one format does not negatively impact the editing experience in an other format. For instance it should not happen that editing a model in a textual format causes the diagrammatic format to lose layout meta-data for parts of the model. To support multi-format editing, the visualizer search algorithm has to be extended to find visualizers in different formats, visualizer metamodels for different formats have to be present and editors which are meta-data preserving (e.g. diagrammatic layout information) have to be supplied.

A mockup of a multi-format modeling environment is shown in Figure 3.3. The left-hand side shows the user-defined company structure modeling language in a diagrammatic editor. The employees are rendered as stickmen. Five employees exist which are Allen, Tim, Jim, Bob and Don. These employees are located in boxes representing the departments they work in. The three departments in this example are Online Marketing, Toy Research and Customer Service. The departments themselves are located in the Quality Toys Inc. company represented by a house pictogram. The editor at the bottom of the multi-format modeling environment shows the same model in a tabular format. Here the employees of the Online Marketing department are displayed

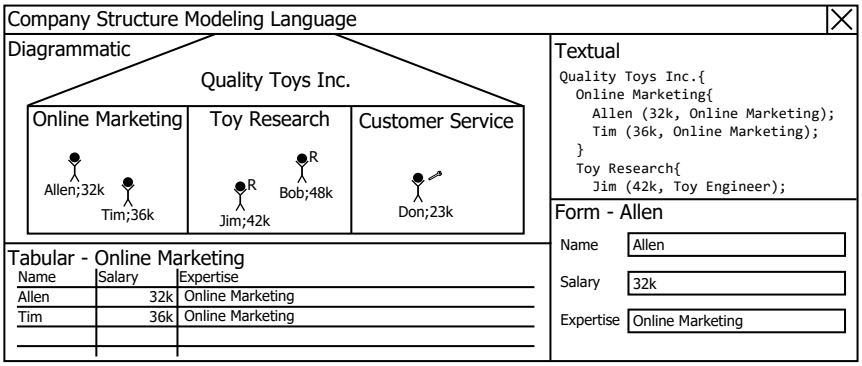


Figure 3.3: A multi-format modeling environment.

in a tabular form. Rows represent the distinct employees and columns their corresponding ontological attributes.

The right-hand side of the modeling environment shows the company structure model opened in a textual format in the upper half and in a form-based format in the bottom half. The textual syntax represents the company by its name and encloses its departments in curly brackets. The departments are represented by their name and enclose their employees within curly brackets. Employees are represented by their name followed by their salary and expertise in round brackets. Employees are terminated with a semicolon. The form-based format shows one selected employee, Allen. The attributes of Allen can be edited through the text boxes holding the respective values.

In a multi-format modeling environment it is possible to use all these views on a model in different formats in an equal way. The modeling workbench does not prefer one format over another and editing in one format does not influence an other format in a negative way. To enable such a multi-format editing experience a projectional approach to language editing is applied throughout all formats including the textual format. This is a derivation from the classical paradigm of editing text through parser-based technologies. In a projectional approach the user directly interacts with the abstract syntax when manipulating a model through the user interface (i.e. its concrete syntax). In a parser-based approach, the concrete syntax is manipulated only through the user interface and then at predefined points in time, e.g. compile or save,

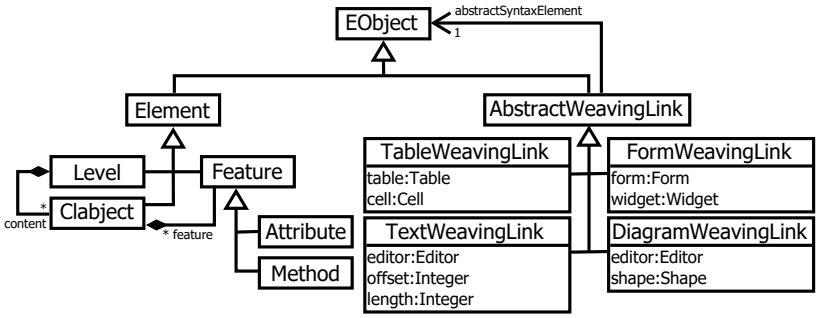


Figure 3.4: Abstraction of the weaving models underlying the projectional editors.

transformed into a model. [235]

Projectional editing is supported by model weaving [35] between the abstract and concrete syntax. This weaving model stores links between an abstract syntax element and its representation in the concrete syntax. A widely distributed example of such a weaving model is the GMF Notation Model [94] based on the OMG Diagram Interchange Specification [179].

Figure 3.4 shows an abstract version of the weaving models created for each format. In this model each AbstractWeavingLink connects an EObject through the abstractSyntaxElement reference from the PLM with its representation in the concrete syntax. By connecting to an EObject it is possible not only to connect a weaving model to instances of metaclasses (e.g. Clabject and Feature) of the PLM but also to their traits (e.g. potency, durability) which are of type EStructuralFeature inheriting from EObject. This ability to support the editing of linguistic traits as well as ontological attributes makes it possible to support format-specific editors leveraging the ontological and linguistic dimension. The subclasses of AbstractWeavingLink have format-specific attributes for weaving an element in the concrete syntax to a model element in the abstract syntax. The TableWeavingLink stores which abstractSyntaxElement is stored in which cell in a table. The TextWeavingLink stores the offset (start position) and the length of the text representing the abstractSyntaxElement in a certain textual editor. A FormWeavingLink stores the widget in a form representing an abstractSyntaxElement and the DiagramWeavingLink stores which shape is representing which abstractSyntaxElement in a certain diagrammatic editor.

When an edit operation is performed in any format-specific editor the weaving link is followed to the abstract syntax and changes are transported from the format-specific editor to the abstract syntax. Each change to the abstract syntax is then immediately transferred to all format-specific editors by following their weaving links from the abstract syntax to the concrete syntax of the changed abstract syntax element. By applying this edit and update procedure all formats can be edited with equal rights by one modeler.

A weaving model needs to support model manipulation operations such as editing, adding, deleting and moving model elements in both the format-specific editor and the abstract syntax model representation of the model being edited in the format-specific editor. These operations, however, are very specific to the format the weaving model is being applied to. Thus, their detailed description is deferred to the subsequent format-specific chapters.

Data: elementToVisualize; udlVisualizer; helper
Result: Check if a visualizer is applicable
1 **if** *udlVislualizer.isInstanceLevel()* **then**
2 | **if** *isSameLevel(elementToVisualize, udlVisualizer)* **then** return false;
3 **end**
4 **if** *helper.isRightFormat(udlVisualizer)* **then** return true;
5 return false;

Algorithm 3.3: Multi-format version of the visualizer search algorithm's applicable function.

The multi-format extension to the visualizer search algorithm's applicable function is shown in Algorithm 3.3. The extension consists of a new format-specific helper input to the applicable function alongside the elementToVisualize and udlVisualizer. This helper is used to configure the visualization search algorithm for each specific format. In this version the helper provides only one function, the isRightFormat()-function. This function takes a visualizer as input and returns a boolean value stating whether the visualizer is applicable for the format of the helper or not. This is usually done by checking whether the visualizer conforms to the user-defined visualizer meta type from the format-specific metamodel for which the helper is implemented. This helper is then used in addition to the original, format unaware algorithm, in line 4 of the

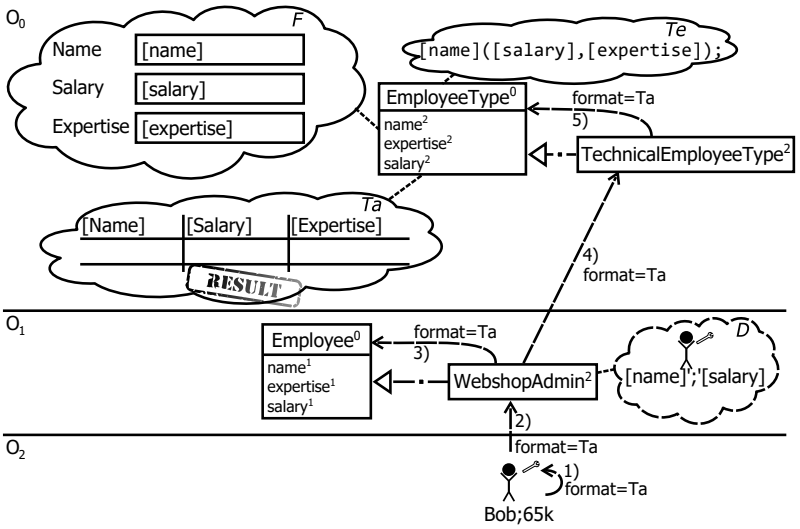


Figure 3.5: A run of the format-aware visualization search algorithm for Bob.

applicable-function as shown in Algorithm 3.3. The extension enables the visualizer search algorithm to search for a format-specific user-defined visualizer with the help of a format-specific helper.

A run of the format-aware visualization search algorithm is demonstrated in Figure 3.5 on the example of Bob. This example shows a subset of the earlier introduced company structure modeling language. User-defined visualizers for four different formats are attached to the clabjects in the example. The format they define the visualization for is indicated by the italic letters at the upper right corner of the clouds representing the visualizers — F for form visualizer, Te for textual visualizer, Ta for tabular visualizer and D for diagrammatic visualizer. Additionally, the arrows representing the trace of the visualizer search algorithm have been extended to show the format for which the search is executed, here format=Ta to indicate a search for a visualizer in the tabular format.

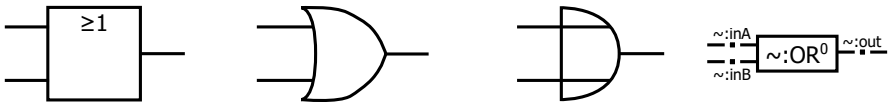
The visualizer search algorithm first looks at Bob for a visualizer but none is found. Next, the supertypes of Bob are searched for a visualizer. This search ends without a result because Bob does not have any supertypes. Afterwards, the types of Bob are searched, WebshopAdmin which does have a visualizer at-

tached. Hence, this visualizer is tested for application by the visualizer search algorithm but the result is negative because the visualizer is for the diagrammatic format and a visualizer for the tabular format is being sought. After this check the supertypes of WebshopAdmin are searched, Employee. Employee does not have a visualizer attached so the search continues at the type level with TechnicalEmployeeType the ontological type of WebshopAdmin. TechnicalEmployeeType does not have a visualizer attached either, so the visualization search algorithm continues at the supertypes of TechnicalEmployeeType, EmployeeType. EmployeeType has three visualizers for three different formats attached — form, text and table. The visualizer search algorithm checks all three visualizers using the applicable function presented in Algorithm 3.3 and returns a positive result for the tabular visualizer. The visualizer search algorithm then terminates and returns the tabular user-defined visualizer.

3.4 Multi-notation Modeling

Until now one user-defined visualization is defined for each format and then used in this format in isolation. Furthermore, a modeler has to decide whether to use the predefined visualization shipped with the deep modeling environment or a particular user-defined visualization. In some situations, however, it can be advantageous to mix and match notations within one format.

Logical gates are an example of a domain in which switching between different user-defined visualizations is beneficial. Three different widely known languages exist to represent the same set of logical gates with exactly the same semantics. Two are the AIEE No 91.-1962 (Figure 3.6(a)) and MIL-STD-806 (Figure 3.6(b)) standards which evolved into today's IEEE Std 91 standard [162]. In addition to these two standards from the American speaking world, the German DIN published the DIN 40700 standard (Figure 3.6(c)). Which standard a domain expert is trained in depends on his cultural background. A domain expert from the English speaking world will more likely be familiar with the ANSI and military notation whereas a person with a German speaking or European background is more likely to be familiar with the DIN standard. Figure 3.6 shows that one concept where the three notations strongly deviate is the *OR-gate*.



(a) (b) (c) (d)

Figure 3.6: Logic *Or*-gate in ANSI (a) military (b), DIN (c) and LML (d) notation.

Multi-notation modeling support within one format can be leveraged to overcome this communication gap between domain experts with different backgrounds talking about the same thing. One domain expert for example can switch the diagram to the ANSI notation when editing while another can use the DIN notation while editing the same model. Thus, every stakeholder on the model has the notation which best fits his/her needs.

A second area of application for multi-notation editing is the ArchiMate syntax of the running example presented here. The running example uses modifications of the *icon notation* of the BusinessActor meta type only. A second notation defined by the standard, however, is a *box notation* in which model elements are distinguished by the shape of the box and the icon placed in the upper right of the box. Multi-notation editing can optimally support such a standard defining two parallel notations.

Multi-notation modeling support can also help a domain novice when using a diagram as the user-defined notation can be exchanged with the predefined notation on-the-fly. Often the classification information displayed by the name compartment of the clabject can give hints about the semantics of a model element when a domain novice does not know the symbol for the concept under question or the name of its ontological type. Such a scenario, where two notations enrich each other in terms of the information they convey, is referred to as symbiotic language support as pointed out in [17].

Using the predefined LML language together with one of the user-defined languages in Figure 3.6, lets a domain novice who does not know the three presented user-defined languages easily find out that the element under question is an instance of OR and does not have a name itself (~). With this information about the type of the model element its exact semantics (e.g. whether

OR represents an *exclusive or* in the language) can be easily looked up in one of the three language's specifications. In addition to information about the model element itself the domain novice can also look up more details about the edges connected to the OR gate. From their type information it is possible to infer that the two edges on the left are the input to the gate and the one edge on the right is the output of the gate.

Multi-notation modeling support can also be used to offer multiple views on one model as shown in [15]. In this example a business process modeling language is enriched with security and performance information. The experts of the different domains, however, do not need all information present when modeling their concern, e.g. performance. Hence, notations for business process modeling, business process performance modeling and business process security modeling can be created to suit the need of each domain expert.

To realize multi-notation modeling support each *AbstractUDLVisualizer* stores the names of all the user-defined notations it is defined for. Each model element can store several *AbstractUDLVisualizers* in the same format providing different visualizations for different notations. Hence, a model element can be visualized in different notations for each format. By contributing to more than one notation, one single visualizer can, for example, contribute to a basic and advanced version of a notation in case it is the same for both. In the context of the company example, a department could refer to the same visualizer for both a basic and advanced version of the notation while employees could refer to different visualizers for the basic and advance versions. For example, the advanced notation could show the salary while basic notation need not.

The default notations in each format are *LML* and *derived*. The *LML* notation uses the default LML predefined visualization whereas the *derived* notation uses the notation defined for the container of the model element to be visualized. By providing the *derived* notation it is possible to switch the notations of whole parts of a model to a certain notation by specifying this notation for a container (e.g. level). *Derived* is the default value for all newly created model elements. Instantiated elements inherit the notation settings from their type as default values but can subsequently display a different notation independent from their type.

The multi-format, multi-notation aware version of the visualization search

```
   Data: elementToVisualize; udlVisualizer; helper
   Result: Check if a visualizer is applicable
 1 if udlVislualizer.isInstanceLevel() then
 2 |   if isSameLevel(elementToVisualize, udlVisualizer) then return false;
 3 end
 4 if helper.isRightFormat(udlVisualizer) then
 5 |   notation ← helper.findNotation(elementToVisualize);
 6 |   if helper.isRightNotation(udlVisualizer, notation) then
 7 |   |   return true;
 8 |   end
 9 end
10 return false;
```

Algorithm 3.4: Multi-format, multi-notation version of the visualizer search algorithm's applicable function.

algorithm's applicable operation is displayed in Algorithm 3.4. The operation expects the elementToVisualize, the udlVisualizer currently under investigation and a format-specific helper as input. The notation to visualize the elementToVisualize is stored in the elementToVisualize as previously described. All modifications making the applicable operation notation aware are highlighted in lines 5 and 6. First, the notation is retrieved by the format-specific helper's findNotation(elementToVisualize) operation in line 5. The findNotation() method returns the notation defined for the elementToVisualize in the helper's format, or in case the elementToVisualize's notation is set to *derived*, traverses the containment hierarchy until a notation which is not *derived* is found. In case no notation other than *derived* is found the findNotation() operation returns the predefined *LML* notation. After retrieving the notation in which the elementToVisualize is visualized, the operation checks whether this notation fits one of the notations of the udlVisualizer currently under investigation in line 6.

Figure 3.7 shows a run of the multi-format, multi-notation aware search algorithm for Jim. The notation in which model elements are visualized is attached as text to model elements using a dotted line, in addition to the visualizers which are attached as clouds to clabjects in the example. The definition of the notation in which a model element is visualized follows the

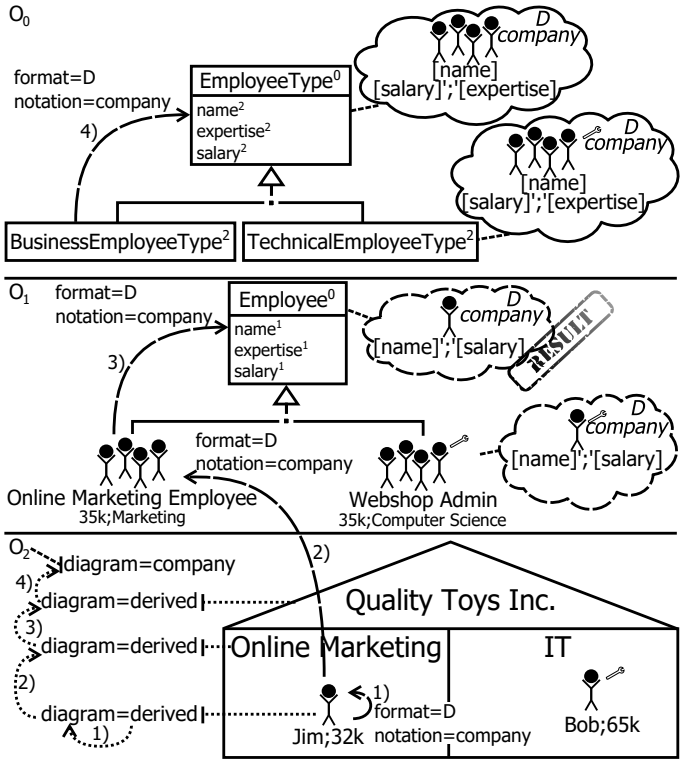


Figure 3.7: A run of the format and notation-aware visualization search algorithm for Jim.

syntax *format=notationName*. The visualizers are extended with all notations they are defined for. These notations are printed in italics below the visualizer's format. In Figure 3.7 only one diagrammatic notation is attached to the model elements and each visualizer is defined for one single notation only. In general, however, as many visualizers and notations in as many formats as needed can be attached to model elements. The dashed arrows indicate the search order of the visualization search algorithm when searching clabjects for an applicable user-defined visualizer, whereas the dotted arrows indicate the search order of the notation search part of the applicable operation presented in Algorithm 3.4. The notation for which a user-defined visualizer is searched is attached to the arrows indicating the search order of the visualization search algorithm in addition to the format.

The visualization search algorithm in Figure 3.7 starts its search with the Jim clabject which does not have a user-defined visualizer attached and does not have any supertypes. Hence, its ontological type Online Marketing Employee is searched for a user-defined visualizer. Online Marketing Employee does not have a visualizer attached either. Its supertype, however, has a visualizer for the diagrammatic format attached as indicated by the D printed in italics in the upper right part of the visualizer. The notation of the visualizer is printed below the format, in this case company. When checking the visualizer for applicability, the notation in which Jim is to be visualized is looked up. The notation of Jim is set to derived so the algorithm uses the notation of Jim's container, the Online Marketing department. Online Marketing has derived set for its notation too. Therefore, the container of Online Marketing is searched. Quality Toys Inc. has derived set for its notation, but its container, level O_2, has the diagram notation set to company. By setting the notation of level O_2 to company all level content which has its notation set to derived is visualized using the company notation, including Jim. Since the algorithm retrieved company as notation of Jim and the visualizer under investigation is applicable for the diagrammatic format and the company notation, the algorithm terminates and returns the visualizer of Employee.

3.5 Aspect-oriented Visualization

The definition of deep modeling languages can span multiple levels as shown in the previous examples. Typically, a general language is defined at the higher levels of abstraction and then refined across the following levels. For instance, in the example used to demonstrate the visualizer search algorithm, very general concepts of workers are defined at the highest level of abstraction, level O_0, and refined by concepts for a specific company in the following level, level O_1. In this example, *web shop administrators* and *online marketing* employees are introduced as company-specific types. The language is then used to model the specific company on the lowest level, O_2. This concept of generic languages which are repeatedly refined across ontological levels is a pattern that can be observed very often in the use of deep models, e.g. in [7, 15, 21, 53, 55, 92, 113, 203]. This pattern of using domain-specific metamod-

eling languages to model other domain-specific modeling languages is referred to as domain-specific metamodeling [242]. De Lara et al. put this term into the context of deep modeling in [55]. In a three level model they refer to O_0 (the most abstract level) as the *domain-specific metamodeling language definition*, O_1 as the *domain-specific modeling language definition* and O_2 (the most concrete level) as the *model*.

When defining such deep languages not only the concepts in the language are refined across classification levels but also their notations. The example in Figure 3.8 shows how the diagrammatic notation for a Webshop Admin is refined across classification levels. Figure 3.8 demonstrates how the notation from EmployeeType represented as a group of stickmen with their name, salary and expertise printed at the bottom evolves to TechnicalEmployeeType, a group of stickmen with a wrench displayed in their upper right, Employee, a single stickman, and Webshop Admin, a stickman with a wrench at its upper right. To define this language using the previously described approach four visualizers would need to be defined which differ only in minor ways as shown in Figure 3.8(a). EmployeeType and TechnicalEmployeeType have basically the same visualizer apart from the wrench in the top right of the symbol. Also Employee and Webshop Admin basically replicate the visualizers of their types for the sole purpose of replacing the group of stickmen with a single stickman.

The problem of defining visualizers in this way is not only that a complementary visualizer has to be defined even when there are only small modifications to the general visualizer provided at a higher level of abstraction. Additional maintenance issues arise when the general part of the notation changes and this change has to be propagated to all visualizers applying modifications manually. By applying aspect-oriented notation definition, as described in [11], these drawbacks can be overcome. The aspect-oriented notation definition feature applies the concepts of aspect-oriented programming languages [130] such as AspectJ [129]. A general notation can be defined and the variable parts can be defined as join points which are configurable through aspects by assigning an identifier (i.e. name) to them. Hence, only modifications to the general notation are modeled and changes to the general notation are automatically transported to all the elements that extend it.

Aspect-oriented notation definition revolves around the concepts of join

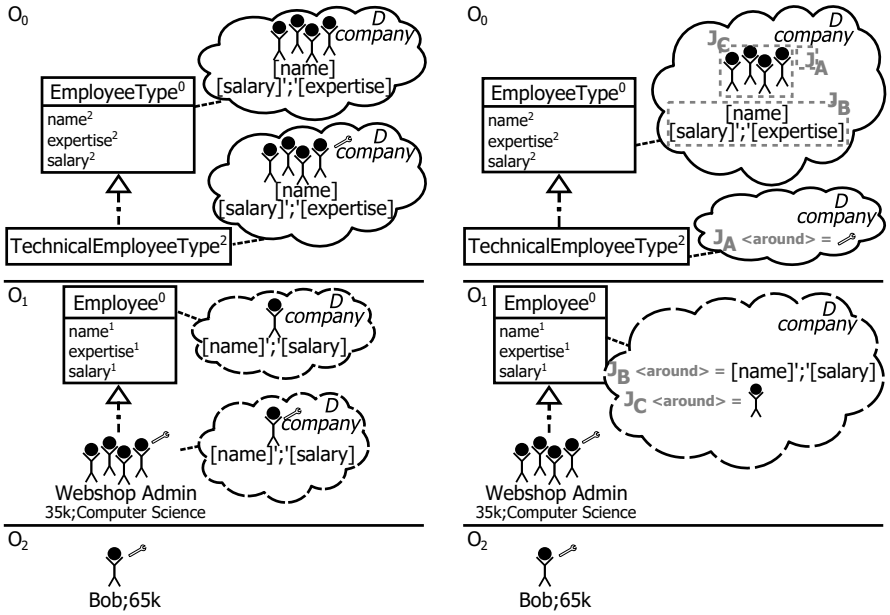


Figure 3.8: Deep notation definition without (a) and with (b) aspects.

points and aspects consisting of point cuts and advices. The parts of a general syntax which are customizable by ontological instances or subtypes in an inheritance hierarchy are declared as join points by assigning them an identifier. Ontological instances and subtypes can then provide aspects for the join points. Three application *kinds* of aspect advices are available: 1. *around*, replacing the join point 2. *after*, placing the aspect content after the join point and 3. *before*, placing the aspect content before the join point. The terms *before* and *after* in the previous description refer to the position in the containment tree relative to the join point in the visualizer which is merged with the provided aspect's advice. In addition to the kind, aspects have a condition which has to hold *true* in order to apply the aspect to a join point. The place of the aspect in the inheritance and classification hierarchies, the condition and the identifier of the join point together build the so-called point cut identifying the join points to which an aspect contributes. The advice of the aspect is the contributed concrete syntax.

Figure 3.8(b) shows the same example as Figure 3.8(a) but with aspect-oriented user-defined notation applied. The diagrammatic visualizer attached to EmployeeType defines three join points for which aspects can provide content. These are J_A to allow an icon to be placed next to the model element's icon (stickman figure), J_B to allow the text shown below the model element's icon to be customized and J_C to allow the model element's icon to be customized. This general visualizer is then customized by TechnicalEmployeeType to display a wrench in the upper right of the model element's icon, where J_A is placed. The abstract Employee clabject further customizes this generic syntax to show a single stickman instead of a group of stickmen (J_C) and to limit the text below the icon to the name and salary of the employee.

A comparison of Figure 3.8(a) and Figure 3.8(b) shows that the version using aspects features one visualizer less than the version not using aspects. Additionally the aspects version does not duplicate the visualizers but only models modifications. This leads to smaller, less complex visualizers and modifications made to the general syntax definition of EmployeeType are automatically transported to all subtypes and instances. We argue that this reduction in the number of model elements needed to model the notation reduces accidental complexity [38]. The positive impact of aspect-oriented notation definition is demonstrated in two case studies [11, 21].

The modified visualizer search algorithm supporting aspect-oriented notation definition is shown in Algorithm 3.5. The search algorithm is modified to check whether discovered visualizers provide a visualizer containing aspects or a visualizer providing a full visualization definition. Aspects are collected until a full visualizer is found and then merged into this visualizer. To implement this behavior a map, name2aspect, is introduced to the algorithm in line 1. This map stores the name of the join point and a list of all aspects provided for this join point. When merging the aspects into the final visualizer different strategies are applied depending on the kind of aspect. Aspects of kind *before* are applied so that aspects defined at more concrete classification and inheritance levels are merged first, whereas aspects of kind *after* are applied in the reverse order. In the case of aspects of kind *around*, only the aspect defined at the most concrete classification level for which the condition holds is applied. In addition to the mapping of join points to aspects, the behavior

Data: elementToVisualize; helper; notation; mergeAspects
Result: A visualizer suitable for rendering elementToVisualize

```
1  name2aspect;
2  types ← elementToVisualize;
3  classification: while (type ← types.poll()) ≠ null do
4      types ← types ∪ type.getDirectTypes();
5      superTypes ← type ∪ type.getDirectSupertypes();
6      inheritance: while (clabject ← superTypes.poll()) ≠ null do
7          for udlVisualizer ∈ getUDLVisualizers(clabject) do
8              if applicable(elementToVisualize, udlVisualizer, helper) then
9                  aspects ← getAspects(udlVisualizer, elementToVisualize,
                     helper);
10                 if aspects = ∅ then
11                     visualizerToMerge ← udlVisualizer;
12                     break classification;
13                 end
14                 addActiveAspectsToMap(udlVisualizer,
                     elementToVisualize, name2aspect, helper);
15             end
16         end
17         superTypes ← superTypes ∪ clabject.getDirectSupertypes();
18         types ← types ∪ clabject.getDirectTypes();
19     end
20 end
21 return mergeVisualizer(visualizerToMerge, name2aspect, mergeAspects,
     helper);
```

Algorithm 3.5: The aspect-aware visualizer search algorithm.

of the visualizer search algorithm is modified in case a visualizer applicable to the notation is discovered (line 9 - 14).

First, the aspects are retrieved from the udlVisualizer using the getAspects operation and stored in the aspects list in line 9. The helper provided to the getAspects operation identifies the format-specific meta types etc. for aspect orientation. Moving this functionality into format-specific helpers allows aspect-orientation to be implemented in a way that optimally fits a specific format. Then the aspects list is checked to determine whether the udlVisualizer provides aspects or not by checking if the aspects list is empty (line 10). If this list is empty the visualizer is stored as the visualizer into which all so far discovered aspects are merged (visualizerToMerge) in line 11, and the search is stopped in line 12. If the udlVisualizer provides aspects, these are stored in the name2aspect map using the addActiveAspectsToMap operation in line 14. This operation checks each aspect of the udlVisualizer whether the condition holds *true* in the context of elementToVisualize and adds it to the name2aspects map. The helper passed to the function, again, handles format specificities in the notation definition metamodels. The final modification to the visualizer search algorithm is the last line, line 21, in which the merged visualizer is returned.

In line 21 the aspect-aware visualization search algorithm terminates returning a visualizer which is merged with its aspects by the mergeVisualizer operation. This operation merges the udlVisualizer with aspects provided via the name2aspects map for its join points in case that it contains entries and the mergeAspects flag is set to *true*. Otherwise the visualizer is not merged and returned by the mergeVisualizer operation.

A run of the aspect-aware visualizer search algorithm as described in Algorithm 3.5 is shown in Figure 3.9 on the example of Bob. Again, the dashed arrows show the traces of the search algorithm. Join points are declared by dashed gray rectangles with a name attached to their border. For instance, the gray rectangle around the group of stickmen in the visualizer of EmployeeType is declared as a join point named J_C. Aspects are defined in visualizers by first stating the join point name, its type (i.e. around, after, before) and then the content (i.e. advice) of the aspect. The visualizer of Employee, for example, provides an aspect J_C replacing (around) the group of stickmen defined in EmployeeType with a single stickman. Visualizers providing aspects are attached

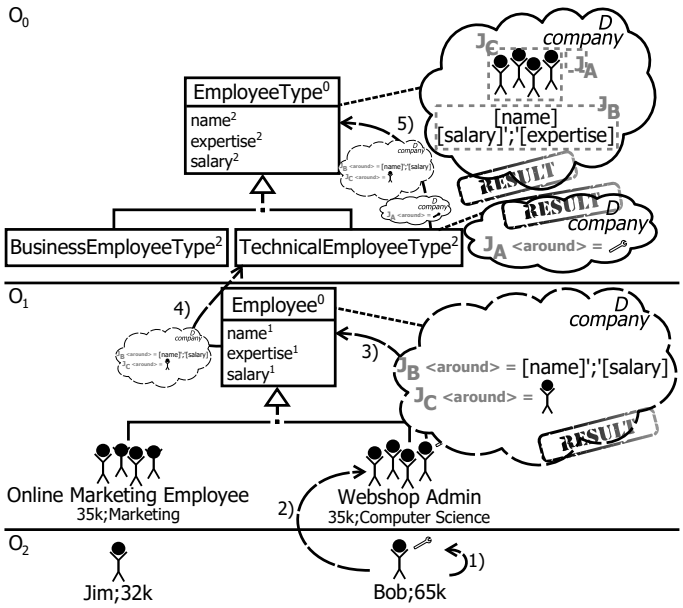


Figure 3.9: A run of the aspect-aware visualization search algorithm for Bob.

to the arrows indicating the visualizer search algorithm trace when collected for future merging by the algorithm.

The aspect-aware visualizer search algorithm trace for Bob shows that first Bob itself is checked for a visualizer. Since Bob does not have any visualizer attached and no supertypes, the ontological types of Bob are searched. Webshop Admin does not have any visualizers attached but its supertype, Employee does. The visualizer provides two aspects: one for join pointJ_B, setting the text displayed for the Employee to its name and salary; and one for join point J_C, setting the figure representing the Employee to a stickman. These two aspects are collected by the visualization search algorithm and the algorithm continues searching in the ontological type hierarchy of Bob, which first contains TechnicalEmployeeType and finally EmployeeType. TechnicalEmployeeType has a visualizer attached providing an aspect for join point J_A which is an icon indicating that Bob is a technical employee. The visualizer search algorithm collects this aspect, too and continues its search at EmployeeType.

EmployeeType specifies a full visualizer which describes a visualization of the model element under investigation and does not provide any aspects. Hence,

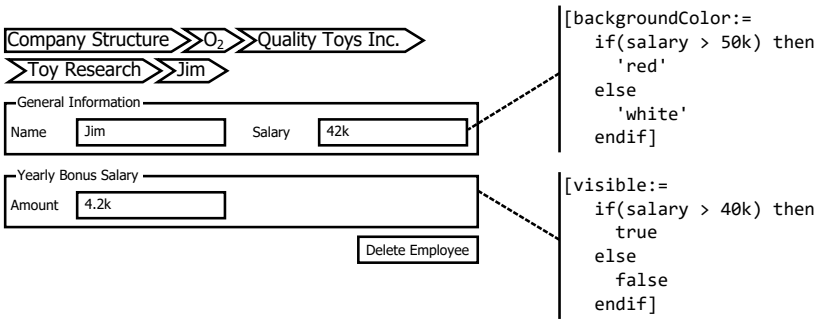


Figure 3.10: The context-sensitive form-based visualization of Jim.

this visualizer is used for the visualization and merged with the collected aspects. In this case, all aspects are of kind around replacing the join points in the visualizer description of EmployeeType. The visualization created by the aspect-aware visualization search algorithm is a stickman with a wrench at the upper right-hand side and the values of the ontological name and salary attributes below it.

3.6 Context-sensitive Visualization

Context-sensitive visualization is a feature frequently required in the domain of model execution as described in [22]. *Context-sensitive* refers to the state of a model at the point of time a visualization is requested. This state is represented by the values of the ontological attributes of the model elements. Examples for context-sensitive visualization are the red coloring of a bottle neck in a business process, or the coloring of attributes when passing a certain threshold.

To realize context-sensitive visualization the notation definitions have to be enriched with constraint expressions. These expressions can for example set attributes of visualizer content, show/hide subtrees of a visualizer or change the color of notation elements.

Figure 3.10 shows a form-based visualization of Jim at its left-hand side. The right-hand side of the figure shows a context-sensitive visualization definition attached to the salary text box and the Yearly Bonus Salary group. The

definition of the context-sensitivity first shows the attribute to set and then the expression calculating the value. In the example, the Salary text box gets a red background color in case the Salary of Jim is bigger than 50k. Also the Yearly Bonus Salary group becomes visible for employees which have a yearly salary higher than 40k and thus qualify for a bonus.

Chapter 4

The Diagrammatic Format

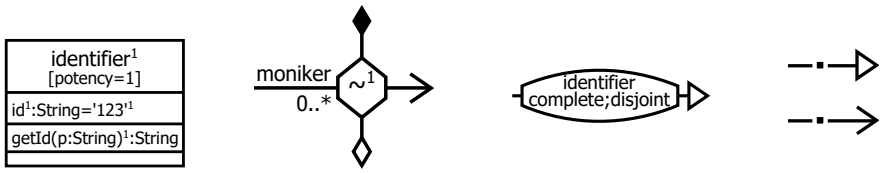
Larkin and Simon [153] describe a diagram as a data structure in which information is indexed in a two-dimensional space. Moody [170] lists lines, graphic areas, 3D graphic elements, labels and spatial relationships as the graphical symbols which can be used in a diagrammatic notation. This thesis does not consider 3D graphic elements, but they could be added by extending the metamodel for defining diagrammatic concrete syntax with concepts from the 3D space. The layout of a diagram (i.e. the location of diagram content in the two-dimensional space) is referred to as secondary notation in [192] which can transport useful additional information in addition to the elements explicitly expressed in the diagram. This information can play a key role in the readability of a diagram. To demonstrate this, the authors of [192] show how bad layout can lead to misunderstandings of diagrams in the electric circuit domain. Störrle [214, 215] also shows a correlation between layout quality and understanding of UML models. Moreover, Klauske and Dziobek [131] argue that their research and experience shows that in Simulink [241] up to 30% of modeling time is spent on diagram layout.

The key advantage of the diagrammatic format over formats presented later in this thesis is its effectiveness in communicating relationships between model elements using spatial layout information and the visual representation of dependencies through edges. The other formats (i.e. text, table, forms) do not have such a feature for graphically representing relationships using edges and spatial information.

4.1 Diagrammatic Predefined Visualization

The predefined diagrammatic visualization for deep models is shown in Figure 4.1. The visual language used as predefined language is the LML as presented in [27, 89, 127]. The LML reuses as many concepts as possible from established modeling languages such as the *Entity-Relationship* (ER) modeling language [45] and the UML [182] but extends the concrete syntax of these languages in a level-agnostic way. The main goal of the LML is to use a uniform visualization for the type and instance facet of a concept in the problem domain. This stands in clear contrast to the UML [182] which uses different visualizations for model elements depending on whether their type or instance facet is represented in a model. In the UML, types are represented by the concrete syntax for classes while instances are represented by a modification of the class' concrete syntax referred to as instance specifications. Other examples of this distinction in the UML are attribute (type level) and slots (instance level) or association (type level) and link (instance level).

The LML has three basic concepts: 1. entities with their attributes and methods (Figure 4.1(a)), 2. connections with attributes, methods and associated connection ends (Figure 4.1(b)) and 3. generalizations (Figure 4.1(c)). Entities are visualized using the widely known UML concrete syntax for classes. The identifier of an entity is a string positioned at the top of the box representing the clabject. Optionally, instead of a simple name, complex statements indicating inheritance, location and classification hierarchies can be specified using deep model element designation as presented in [10]. Next to the identifier the potency of the clabject is displayed using the superscript (potency) notation. Below the identifier, a list of linguistic attributes and their values can be displayed within square brackets, for example, to make trait values more explicit to a modeler. This section is referred to as the *trait value specification* (tvs) based on the name given to linguistic attributes — traits. The first compartment of an entity displays its attributes. In Figure 4.1(a) the example attribute is called id. The name of the attribute is followed by its durability, 1 in the example, its datatype if defined (here String), its value (here '123') and the mutability displayed next to the value using potency notation, 1 in the example. In order to promote readability and reduce the amount of infor-



(a) (b) (c) (d)

Figure 4.1: Visual concepts of the level-agnostic modeling language.

mation displayed in the LML, elision rules can be applied to hide information which can be derived from the model context. For example, durability of an attribute can be hidden if it is the same as the containing clabject's potency and the mutability can be hidden if it is equal to the attribute's durability.

The second compartment of an entity contains the operations describing the behavior of the entity. In the example an operation named getId is contained by the entity. An operation's parameters are displayed in round brackets. They are defined through their name followed by a colon and the indication of their data type. In the example one parameter named p of type String is defined for the operation getId. The parameter definition is followed by the operation's durability in potency notation followed by a colon and the return type of the operation, here String. Again elision rules are applied to the durability of operations. In case that the durability is equal to the clabject potency it is hidden.

Model elements which are owned by an entity are represented in the final compartment which is blank in Figure 4.1(a). These can be entities, connections and inheritance relationships. Again, elision rules can be applied to hide empty compartments if they do not contain any information.

The notation for connections is inspired by the ER notation in which relationships are represented as diamonds to which attributes can be attached in the form of ellipses. The UML supports this diamond notation for associations, too, but it is usually only used for higher-order associations. In the UML, association's can be given features using the association class concept. Association classes are presented using a class symbol connected to the association via a dashed line. In LML all connections are essentially association classes, in contrast to the UML which distinguishes between associations and association classes.

Figure 4.1(b) shows a connection rendered in the LML. This connection has no name given (∼) and a potency of 1. The example connection does not own any attributes, operations or model elements and, hence, the corresponding compartments can be hidden for space reasons. Attributes, operations and owned model elements are visualized in the same way as within entities shown in Figure 4.1(a). A connection is connected with other clabjects (i.e. entities and connections) via connection ends, represented as solid lines. Four kinds of connection ends exist: 1. composition (filled diamond line decoration), 2. aggregation (not filled diamond line decoration), 3. navigable (arrow line decoration), and 4. not navigable (no line decoration). Monikers naming the connection ends and multiplicities are attached as strings to the connection end that they name or constrain as seen in Figure 4.1(b). A detailed description of connections supported in LML is found in [20, 97].

Generalizations (Figure 4.1(c)) are visualized as rectangles with a curved top and bottom. An identifier can be placed within the shape together with statements about properties of the associated generalization sets — complete, incomplete, disjoint, overlapping. These properties cover the same semantics as in the UML [182] and are, hence, not further described here. A generalization can be connected with an unlimited number of super- and subtypes. A supertype is indicated by a line decorated with a non-filled triangle while a subtype ending has no line end decoration.

Connections and generalizations can alternatively be represented in an imploded form as shown in Figure 4.1(d). In this form the node in the middle of the connection is replaced by a small black rectangle about the size of the lines it is connected with. This feature called *dotability* [127] is very useful for reducing the space used by generalizations and connections which do not provide additional information (e.g. attributes) to be rendered inside the exploded form.

4.2 Diagrammatic User-defined Visualization

The metamodel for defining the concrete syntax of a modeling language in the diagrammatic format is based on the notions of layout, shapes and labels like most metamodels for describing diagrammatic formats, e.g. [71, 94, 102, 175].

With these concepts all ingredients of a diagrammatic modeling language as earlier enumerated and defined by Moody in [170] can be modeled.

Figure 4.2 shows the metamodel used to describe visualizers for the diagrammatic format. The DiagramVisualizer is the root for the visualizer description. It consists of instances of the abstract subclasses of VisualizationDescriptor which are LayoutDescriptor, describing the layout of content within shapes, and LayoutContentDescriptor, describing the shapes and labels placed in layouts. Three different layouts exist, FlowLayout, TableLayout and AbsoluteLayout. The FlowLayout aligns its content either horizontally (vertical = *false*) or vertically (vertical = *true*). The content is aligned in one row until the border of the containing shape would be exceeded. The content exceeding the border is then placed on the next row. The FlowLayout can be forced to be single-lined by setting its singleLine attribute to *true*. The TableLayout organizes its content in a grid where the number of columns is determined by the columns attribute. Each piece of content is placed in its own column. When the number of columns is exceeded, the content is placed on the next row. The AbsoluteLayout allows its content to be freely positioned within the layout containing shape. For all three forms of layout, the margin and padding can be configured through the Margin and Padding types subclassing SpacingDescriptor which provides x and y coordinates to describe spacings.

The different types of LayoutContentDescriptor contained in a LayoutDescriptor through its content attribute are ConditionalLayoutContent and RenderedLayoutContentDescriptor. ConditionalLayoutContent can be used to show or hide a concrete syntax element depending on the condition attribute, which is inherited from ConditionalDescriptor. If the condition evaluates to *true* the content is shown, otherwise it is hidden. RenderedLayoutContent is the superclass for all content in a layout which is intended to be viewed by a model user. Its backgroundColor and foregroundColor can be set to either a user-defined color specified as an RGB-value using the RGBColor class or to a predefined color using the StandardColor class. In the current implementation, the predefined colors are Black, White, Blue, Red, Green, Yellow and Orange but these can be extended as needed.

The RenderedLayoutContent's subclasses ShapeDescriptor and InformationDisplayDescriptor are the shapes available in the diagrammatic format to visualize

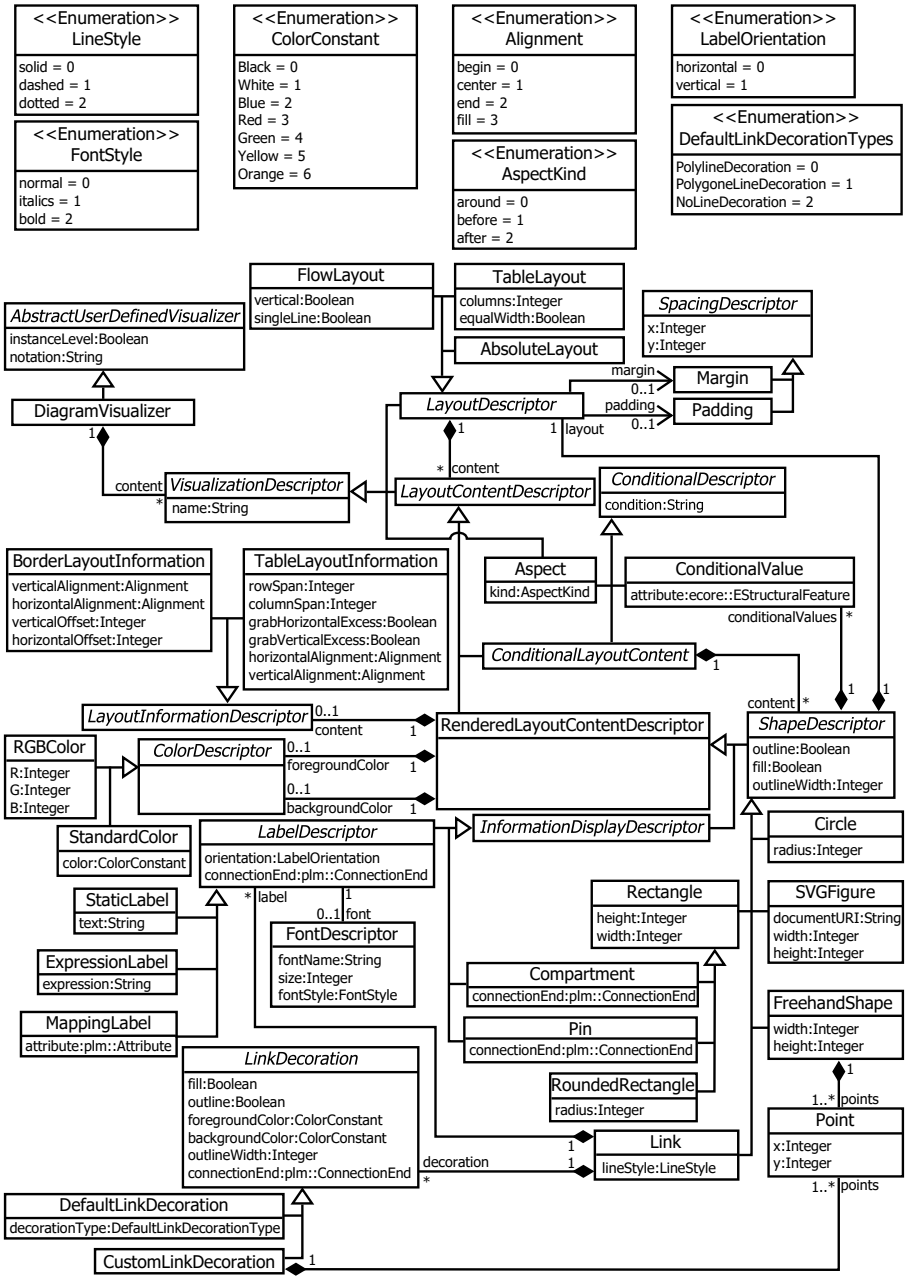


Figure 4.2: Diagrammatic visualizer metamodel.

model content. The geometric shape (e.g. rectangle, circle) of a visualized model element is modeled by subclasses of ShapeDescriptor. Each geometric shape contains a LayoutDescriptor which determines how content visualized within the shape is arranged. The visualization of a ShapeDescriptor can be influenced by the outline attribute, determining whether or not to draw the border of the shape, the fill attribute, determining whether the background of a shape is filled with a solid color (fill = *true*) or is transparent (fill = *false*), and the outlineWidth attribute specifying the thickness of the shape's border. The geometric shapes available in the diagrammatic format are Circle, SVGFigure, FreehandShape and Rectangle with its subclasses Compartment, Pin and RoundedRectangle. The radius of a Circle can be specified through its radius attribute, and the width and height attributes of SVGFigure, FreehandShape and Rectangle influence the visual size properties of the corresponding shapes. RoundedRectangle is a special kind of Rectangle which has rounded corners defined through the radius attribute. Compartments are a kind of rectangle which are place holders for abstract syntax elements owned by the model element to be visualized. A well known example of this are UML classes which store attributes in their attribute compartment. The abstract syntax instances to be stored in the compartment are specified through the connectionEnd attribute. In some graphical languages, connections to a model element end in rectangles which are placed at the border of the visualized model element, e.g. ports in the UML. This is modeled through Pins which define the model elements to which they are connected through the connectionEnd attribute.

For some languages the default geometric shapes provided by the visualizer definition metamodel are not sufficient. Two alternatives are offered to address this, FreehandShape and SVGFigure. The shape of a FreehandShape is defined through Points defined by x and y coordinates. These points are then connected by straight lines. More complex shapes are possible by employing the SVGFigure, which uses an image in the Scalable Vector Graphic (SVG) format provided at the documentURI location for visualization.

Information from the model stored in the abstract syntax is displayed through InformationDisplayDescriptors with their subclasses LabelDescriptor, and the previously presented Compartment and Pin. A LabelDescriptor can be configured to display its text either horizontally or vertically depending on the value

of its orientation attribute. The font is configured by applying a FontDescriptor which defines a fontName, size and fontStyle (normal, italics and bold). Three kinds of labels exist: 1. StaticLabel, displaying non-changeable, predefined text, 2. ExpressionLabel, calculating its displayed text based on the result of the expression defined in the expression attribute, and 3. MappingLabel, displaying the value of a specified attribute.

The attributes presented here for determining the visualization of a ShapeDescriptor do not have to be statically set at design time but can be calculated during run time to support context-sensitive visualization. This is supported by the ConditionalValue type connected to ShapeDescriptors through the conditionalValues reference. For each attribute of a ShapeDescriptor a condition can be defined. The result of this condition is then used to set the attribute's value at run time. For instance, using this feature the outlineWidth of a ShapeDescriptor can be varied based on attribute values stored in the visualized model.

To refine the alignment of a shape within its container, RenderedLayoutContentDescriptors can provide a LayoutInformationDescriptor. These are available for shapes within a table layout (TableLayoutInformation) and shapes displayed outside the border of another shape (BorderLayoutInformation). A TableLayoutInformation can configure a shape to span a certain number of columns (columnSpan), a certain number of rows (rowSpan), fill empty horizontal space (grabHorizontalExcess), fill empty vertical space (grabVerticalExcess), define their verticalAlignment and horizontalAlignment. The options for these alignments are begin, center, end and fill.

Connections between model elements are visualized through Links. The visualization of the line of the connection is defined by the lineStyle attribute which offers solid, dashed and dotted line styles. Labels can be attached to the link by using LabelDescriptors via the label reference. The visualization of the Link ends is specified by LinkDecorations via the decoration reference. LinkDecorations can be set up to fill the background with a solid color or display a transparent background, display a border (outline), have a certain foregroundColor, backgroundColor, outlineWidth and be associated with a certain connectionEnd of the Link. Two kinds of LinkDecorations are available: DefaultLinkDecoration being visualized as one of the three decorationTypes — PolylineDecoration, PolygoneLineDecoration and NoLineDecoration. CustomLinkDecorations are used to specify

the link decoration as a set of Points via their x and y coordinates.

Parts of the diagrammatic visualization definition can be made addressable as join points which can be further refined by aspects. To do so their name attribute inherited from the VisualizationDescriptor is set to a unique value. Aspects are then used to provide visualizations, modeled through their content attribute, for these join points. An Aspect is applied only when the condition attribute inherited from ConditionalDescriptor is *true*. Three application kinds for Aspects are available as earlier described: 1. around, replacing the join point with the Aspect content, 2. before, placing the content of the Aspect before the join point and 3. after, placing the content of the Aspect behind the join point.

4.2.1 Diagrammatic Visualizer Metamodel Example

An instance of the diagrammatic visualizer definition metamodel is shown at the top of Figure 4.3. The definition of the visualizer is shown as a containment tree where the text at each node determines the type of the model element and the attributes and their values are specified in brackets. The visualizer shown in Figure 4.3 defines the diagrammatic visualization of employees in the company structure modeling language example.

The shape consists of a Rectangle surrounding the whole visualization of the employee. This rectangle has a transparent background (fill=false) and does not show a border (outline=false). The outer rectangle uses a TableLayout with one column (columns=1) to arrange its content. First an SVGFigure is placed into this Rectangle to display an icon of the visualized model element. Second, a Rectangle containing labels displaying information from the employee's ontological attributes is placed in the outer Rectangle. The SVGFigure has TableLayoutInformation attached to occupy all available horizontal space (grabHorizontalExcess=true) and all available vertical space (grabVerticalExcess=true). Furthermore, it fills its container horizontally and vertically (horizontalAlignment=true and verticalAlignment=true). The displayed icon is made context-sensitive by the use of a ConditionalValue which sets the documentURI of the SVGFigure (attribute=documentURI). The documentURI is either researcher.svg, a stickman with an R at the top right, or employee.svg, a plain stickman, based on the value of the employee's researcher attribute as the condition shows.

The rectangle containing the labels displaying the ontological attributes of

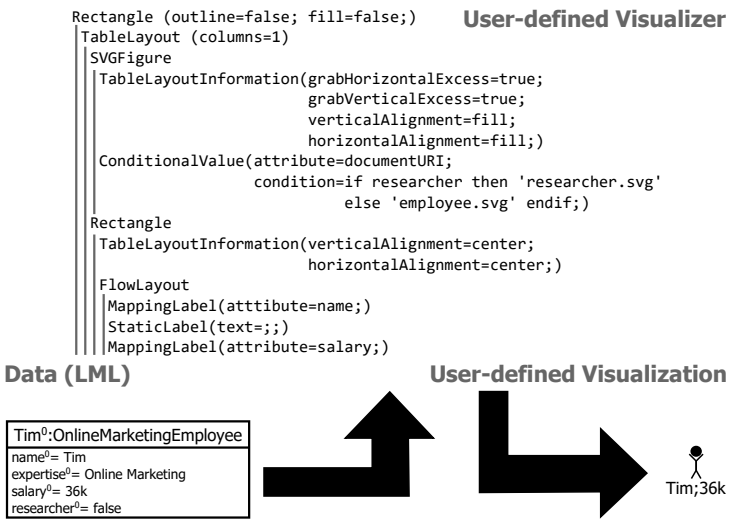


Figure 4.3: Tim visualized in the diagrammatic company structure modeling language.

the employee has a TableLayoutInformation attached which configures it to be placed horizontally centered (horizontalAlignment=true) and vertically centered (verticalAlignment=true) in its layout cell. The Rectangle uses a FlowLayout to arrange its contents. The first label is a MappingLabel displaying the value of the name attribute (attribute=name), the second label is a StaticLabel displaying the static non-changeable text *;* (text=;) and the third label is a MappingLabel displaying the value of the salary attribute (attribute=salary).

The bottom left of Figure 4.3 shows Tim in the LML notation to which the user-defined visualizer is applied. The result of the visualizer application is shown in the bottom right, which is a stickman pictogram with the text Tim; 36k placed below it.

4.3 Diagrammatic Weaving Model

The weaving model used to keep the model represented in the diagrammatic format synchronized with its abstract syntax model representation is displayed in Figure 4.4. The model is identical to the *Notation Model* [94] used in the GMF run time project to connect diagrammatic editors with their abstract

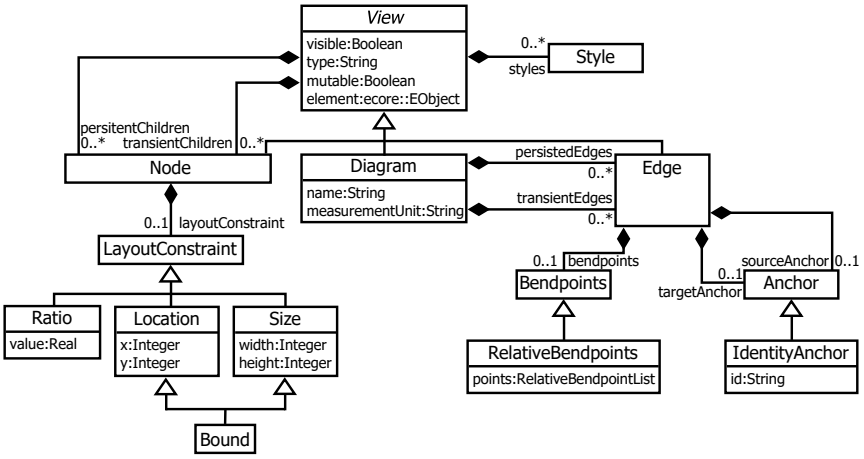


Figure 4.4: Diagrammatic weaving model (GMF Notation Model after [94]).

syntax. The central type in this metamodel is View which is the superclass for Node, Edge and Diagram. A view is connected to the abstract syntax of the model via the element attribute. The diagram editor uses this attribute to synchronize changes between diagrammatic concrete syntax representations and the abstract syntax. Besides establishing links between the concrete syntax and abstract syntax, the metamodel is also used for persisting layout information about a diagram. For instance, the visible attribute of View stores whether a diagrammatic representation of an abstract syntax element is rendered or not and the Size type stores the width and height of a diagram node. The layout purpose of the weaving model is not discussed further here, however, since it duplicates the information stored in the LML visualizers.

The weaving model operations for the diagrammatic format are not explained in detail here since they are rather trivial to realize. This is because the add, remove, move and edit operations do not break any layout that would cause recalculations for the not edited parts in the weaving model. Furthermore, both the abstract syntax model representation and the representation in the format-specific editors can easily be addressed via pointers. When a new element is added either via the abstract syntax model representation or the format-specific editor, a new View corresponding to the visual type of the added model element (i.e. Edge, Node) is added to the weaving model and the

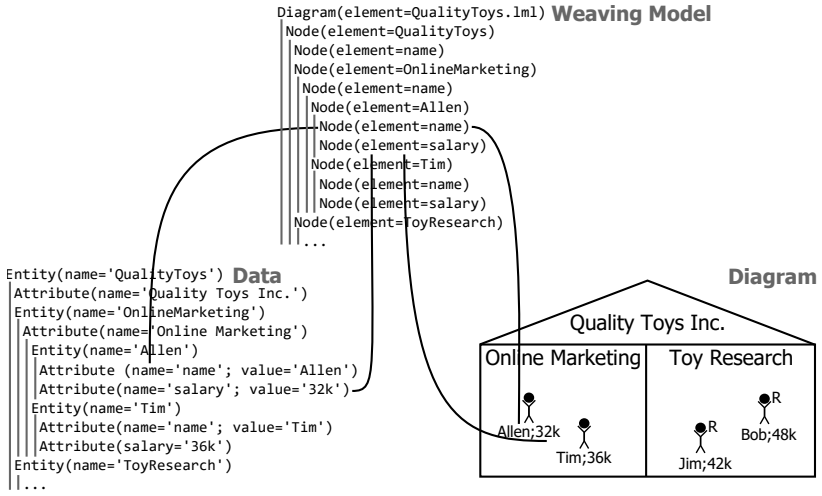


Figure 4.5: Diagrammatic weaving model on the company structure modeling language example.

other side is updated correspondingly. The deletion operation is similar with the only difference being that the deleted model's view is removed from the weaving model and, hence, is removed from the abstract syntax model representation and the format-specific editor. When a model change occurs, the edited element's view is looked up in the weaving model and the format-specific editor and abstract syntax model are updated.

4.3.1 Diagrammatic Weaving Model Example

Figure 4.5 shows an example instance of the diagrammatic weaving model for the company structure modeling language. The right-hand side shows a company producing toys, called Quality Toys Inc., visualized in a diagrammatic user-defined language as a house. The company consists of two departments, Online Marketing and Toy Research indicated by rectangles within the company. These departments contain their employees which are represented by stickmen. In addition, research staff have an R displayed in their upper right corner. Below the stickman symbol the name and salary of the employee are rendered. Four employees exist — Allen, Tim, Jim and Bob.

The left-hand side of Figure 4.5 shows the company structure in its abstract

syntax representation visualized as a tree. For each entity and attribute a node is created in the tree with the details of the represented entity in brackets. The abstract syntax representation and user-defined diagrammatic representation are connected through the weaving model in the top center of Figure 4.5. This weaving model is a cut down version of the model a GMF editor would produce since for example no layout information such as x and y coordinates is stored. For each entity in the diagrammatic and abstract syntax representations of the model, one Node is stored in the weaving model. These Nodes point to the abstract syntax and diagrammatic representations. In the example, two weaving links, for the name and salary attributes of Allen, are highlighted by solid lines. The fact that weaving links are realized as pointers and that model elements do not influence each other's information in the weaving model makes it easy to implement model manipulation operations on top of the diagrammatic weaving model.

Chapter 5

The Textual Format

In contrast to diagrams which encode information in a spatial way, text is classified as sentential by Larkin and Simon [153], meaning that it is a sequential format. The lack of secondary notation (diagram layout) in models represented as text is described as an advantage over diagrammatic representations in [192] because the layout of a diagram can in many situations lead to misinterpretations of the model. In a sentential format, text used to represent domain concepts and their relations preserves logical and temporal sequences in contrast to the diagrammatic format which focuses on the spatial relationships between model elements [153].

The advantage of text over the other formats presented in this thesis is that it is a very efficient way to enter lots of new data into a model. A modeler does not have to interact with any mouse-driven user interface or care about layout, etc. All that a user has to do is to type in text using the keyboard. Modern textual modeling environments raise productivity even more in the form of proven assistance mechanisms (e.g. automatic correction, code completion, code snippets) or new research trends such as keyword programming [157], semantic auto completion [112] and example-based code completion [40]. On the other hand, to efficiently use text for inputting a model, a certain degree of expertise in the applied textual modeling language is needed, because users do not received as much guidance from the modeling environment's user interface as in other formats.

(a) (b)

Figure 5.1: A user manipulating a model through a free text modeling tool (a) and a projectional modeling tool (b).

5.1 Textual Editing Paradigms

In [235] two approaches for editing models in a textual format are described — the widely applied free-text modeling tools (Figure 5.1(a)) supported by a parser and command-directed, projectional modeling tools [80] (Figure 5.1(b)) which directly edit the text's underlying abstract syntax model through a textual projection. Text-based, projectional modeling tools have their origin in syntax-directed editing tools [159] from the 1970s/80s such as Emily [99, 98], Mentor [149] or the Cornell Program Synthesizer [220]. In the following subsections the advantages and disadvantages of these two approaches are outlined.

5.1.1 Free-text Model Editing

When applying free-text model editing, a user can write any text without any restrictions into a text editor. This plain text is then converted into a model often referred to as the abstract syntax tree (AST) of the entered text. The abstract syntax tree is the representation of the text which is processable by a machine. The step from the plain text to the abstract syntax tree is called parsing [3]. Parsers can be either implemented manually or generated using *grammarware* tools based on a user-defined grammar. In the latter case a parser generated automatically from the user-defined grammar recognizes text according to its underlying context free grammar specification in e.g. EBNF [239] and creates the corresponding abstract syntax tree. The clear advantage of the free-text editing approach is that a modeler can freely type any text and save the file at any state no matter whether the text conforms to the specified grammar. This enables scenarios such as copying model snippets from any

Figure 5.2: The dangling else problem after [1].

external source (e.g. a website) and reusing them in a model. On the other hand, the parser-based approach has several drawbacks when it comes to the multi-format editing scenario proposed in this thesis — namely, the level of expertise needed to define a textual language, the restrictions placed on the defined language and merging issues.

First, mechanisms and rules to handle ambiguity [42] in the defined language have to be employed. Ambiguity in a language is defined by Brabrand et al. [37] as a "situation where a string may be parsed in multiple ways, leading to different parse trees." One example of ambiguity is the dangling else problem [1] shown in Figure 5.2(a) and Figure 5.2(b) that arises in computer programming languages. Both figures show the same example with different indentation to make the problem clear. Based on the language grammar it is not possible to decide if the else clause belongs to the first if clause or the second if clause. In XText grammar definitions, for example, such ambiguity is resolved by enriching the grammar with syntactic predicates that give hints to the parser. Ambiguities also occur in language composition scenarios where two grammars are designed independently but then combined. Language composition is envisaged as central part of the deep modeling approach used in this thesis as presented in [15] in the context of the composition of deep business process modeling languages.

Second, the language defined using a grammarware tool has to match the category of the generated parser. Example parser categories are LL(*) [190], as supported by ANTLR [189] (and thus also XText which uses the ANTLR3 parser generation framework) and LALR(1), supported by YACC [118]. Even when the language under design is supported by the chosen parser generation framework, the grammar has to be designed so that a parser can be generated for it by the language workbench's underlying parser generation framework.

This requires a language engineer to be aware of many technical details about the underlying parser generation framework, for example how left recursive grammars are handled.

The third, and for multi-format modeling, most significant drawback of free-text model editing is that the primary representation form of the model is unstructured data, and the abstract syntax tree model is only created at the point in time when the model is consumed by a machine as for example described in [78, 205, 206]. This reduces the abstract syntax model underlying the textual representation to being a temporary artifact because modelers and tools focus only on the concrete syntax which is then translated into a model for machine interpretation.

In multi-format editing, however, the textual format is just another format alongside the diagrammatic, tabular and form-based formats and does not play a special role for interaction, computation and persistence. Furthermore, by applying the multi-notation paradigm a certain textual representation of a model is just one of many ways to express the model in a textual form. To support multi-format, multi-notation model editing the model represented as text is saved in a way that is independent of the (textual) format-specific concrete syntax.

Shifting the focus from text as the primary artifact to the underlying model represented in a format-independent way creates problems in situations in which a modeler writes text which does not conform to the concrete syntax. In such a situation, a user cannot save the edited model until it fully conforms to the concrete syntax as it cannot be parsed into a form that can be merged with the model represented through the abstract syntax model underlying all the modeling formats.

Another problem is that multi-format textual editors rely on parsing the free-text into a model and then merging this with the original abstract syntax model underlying all the formats to be edited via the textual format. In situations where the merging algorithm cannot detect changes correctly, such as changes to attributes uniquely identifying model elements, the meta-data of other formats (e.g. diagrammatic layout) can get lost. In this case, the merge algorithm would assume that the modeler intends to delete the model element holding the old value in the identifying attribute and would thus delete it.

Then a new model element is created during the merge operation holding the new identifying value. This newly created model element does not hold the meta-data of the deleted model element and thus the non-textual formats in which the model is also expressed (e.g. the layout of a diagram) are negatively influenced.

Moreover, changes to the underlying model from any format other than the textual one are difficult to merge into the displayed text. A naïve approach for synchronizing changes from all other formats with the textual format is to rewrite the text displayed in the textual editor. This, however, destroys the text formatting created by a modeler and would move the cursor position in the text. An alternative approach is to locally merge changes with their textual representation. This, however, is not trivial due to the missing links between the abstract syntax model element representations and their counterparts in the unstructured textual representation in the textual editor.

5.1.2 Projectional Editing

The drawbacks caused by the combination of free-text editing with the meta-model focused multi-format editing approach have led to interest in an alternative textual model editing paradigm — projectional editing. Projectional editing essentially relies on the same model editing paradigm used to support tabular, diagrammatic and form-based, etc. model editors. In this approach to model editing, the model is stored in a representation independent format, the abstract syntax representation, and no parser is needed to transform the edited text into an abstract syntax model representation. If a modeler manipulates a model through a projectional model editor, the abstract syntax representation is directly changed instead of the concrete syntax representation which is then transformed into the abstract syntax model representation. In other words, the modeler interacts with the model through a textual projection of the abstract syntax representation of a model which is modeling format independent. When editing, a user is directly editing the abstract syntax of the model and not manipulating the concrete syntax which is then translated into the abstract syntax representation of the edited model. By transferring this paradigm to textual modeling the concrete syntax is no longer the primary format to interact with a model and it is reduced to a secondary role. This

solves the merge problem between abstract syntax and text in both directions in a multi-format modeling environment because a modeler always immediately changes the abstract syntax representation underlying all formats of the edited model. It is therefore no longer necessary to merge the model created by a parser from the edited text with the abstract syntax representation underlying all formats of the edited model. The lack of a parser, however, means that no unstructured data (i.e. text) can be imported since model elements can only be created through the projectional text editor. For instance, no model snippets can be copied from e.g. a web page or e-mail into the projectional editor.

Projectional editing not only solves the problem caused by merging in multi-format editing but also the other aforementioned problems of free-text editing. Ambiguity in grammars is no longer a problem since the user always selects what role an element in the textual representation plays. Thus, disambiguation is performed by the user at model creation time and not deferred to the parser once the model content has been created. This frees a language engineer from having to define syntactic predicates to address ambiguities in a grammar. Also since no parser is required, the language engineer does not need to know what the capabilities of the language workbench's underlying parser generator are. A language engineer can create the grammar that best fits the problem in hand rather than the parser generator. In fact, it is possible to create grammars which cannot be parsed at all. This also reduces the complexity of specifying a grammar.

Directly editing the abstract syntax of a model through text, however, places restrictions on how a textual model editor supports model editing. The text in a projectional model editor must at all times fully conform to the concrete syntax defined for representing the data stored in the underlying abstract syntax model. Thus, all model elements must be expressed as a whole in the projectional model editor and cannot be sequentially built up by typing text as in a free-text editing environment. Hence, a user has to use content creation operations offered by the projectional editor to create text representing whole model elements. Examples of such operations are content-assist pop-ups which offer entities to instantiate, connections available for connecting elements and selectable enumeration attribute values etc.

Because of the need for such content creation operations, free-text editors were preferred to projectional editors for many years. In the early days of syntax directed editing, Bernard Lang, for example, stated in [149] that the user interfaces of the tools are too complex for inputting programs or performing simple editing tasks. They, however, see strengths in maintenance operations executed on programs stored in a projectional programming environment. Voelter et al. identify several more drawbacks and classify them in [233]. They then demonstrate that modern projectional editors such as JetBrains MPS can overcome these drawbacks. Amongst other things, JetBrains MPS applies a hybrid projectional / free-text editing approach in which a user can start to write arbitrary text which is over time parsed into the underlying model. This enables scenarios such as copying model snippets represented in plain text into the modeling environment. Another modern projectional editing workbench demonstrating the power of the approach is the Intentional Domain Workbench [115].

In this thesis, the projectional approach to textual modeling was chosen to support multi-format editing since the advantages of directly editing the abstract syntax through text far outweighs the usability drawbacks in this scenario.

5.2 Textual Predefined Language

The textual, predefined language used to visualize deep models is inspired by the MetaDepth language developed by de Lara et al. [50, 167] which is a deep version of the OMG's Human-Usable Textual Notation (HUTN) [178]. The main modifications made to this syntax are the addition of: 1. durability and mutability to attributes, 2. generalization sets and 3. the containment relationship between clabjects. Since the complete integration of MetaDepth was not a goal of this thesis, rather than giving a full specification of its grammar this section provides only a brief example-based introduction to its concrete syntax.

Even though the predefined textual syntax presented here is based on the MetaDepth language there are significant differences between MetaDepth and the LML supported deep modeling approach. For example, MetaDepth

does not support mutability, the definition of methods inside clabjects and declaring clabjects as abstract by setting their potency to zero. Additionally, it has a slightly different classification semantics and features concepts such as leap potency which are not available in the LML modeling approach. A more detailed description of MetaDepth is available in [50] and the differences between the two deep modeling approaches are outlined in [12].

Figure 5.3 shows the additions to MetaDepth's concrete syntax to support the LML deep modeling approach described here. The top half of the figure shows a clabject, identifier, inheriting from (supertype)identifier and connected to (target)identifier using the diagrammatic LML notation. The bottom half of the picture shows the same model but in the MetaDepth inspired language adopted here. The model is visualized using a small pseudo grammar in which static text is written in single quotation marks (“ ’ ”) and linguistic trait values derived from the metamodel are enclosed in square brackets (“[ ]”). A choice between two representations is represented by the pipe symbol (“|”), e.g. “’A’|[B]”. Some linguistic trait identifiers that retrieve data from the model occur more than once in Figure 5.3, hence, they are made unique by putting a string in round brackets before them (“( )”).

As shown in the bottom of Figure 5.3, in MetaDepth a clabject is defined by the keyword Node if it does not have an ontological type followed by its identifier. If a clabject does have one or more ontological types, all (classifier)identifiers are listed in a comma-separated list instead of the Node keyword. The identifier is followed by a colon and the (supertype)identifiers if the clabject has any supertypes. Then the potency of the clabject is defined with a leading @ sign. The contents of a clabject, such as attributes, methods and other clabjects, are enclosed in curly brackets.

Attributes are defined through their (attribute)name followed by the @ sign and their durability followed by a comma followed by their mutability. Then the datatype is separated from the durability and mutability by a colon followed by an equals sign and the value of the attribute. Methods are represented first by their (method)name followed by a pair of round brackets, their potency indicated by a leading @, and their body surrounded by curly brackets. Connections in which a clabject participates are represented through the [(t)moniker] located at the opposite end of the connection, followed by a colon and the name of

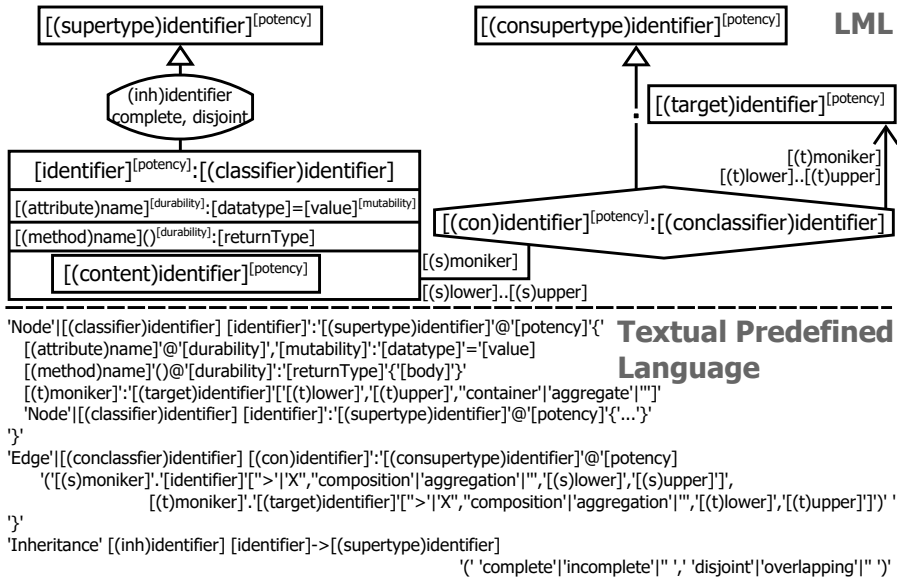


Figure 5.3: LML concepts in the textual predefined language.

the clabject connected via the connection end. The multiplicity is indicated in square brackets using the notation "[(t)lower]','[(t)upper]". Additionally, the keywords *container* and *aggregate* can be defined if the connected clabject is contained or aggregated by the model element. Instances of contained clabjects are rendered within their owning clabject using the same syntax as if they were located in a level.

Connections, which are clabjects connecting two or more other clabjects, are represented through an extended clabject syntax. If they do not have an ontological type their representation starts with the keyword Edge, otherwise it starts with a comma-separated list of (conclassifier)identifiers. Subsequently the (con)identifier is declared followed by the list of (consupertype)identifiers separated by a colon from the (con)identifier. The potency is then defined with a leading @ symbol. Finally, the ends of the connection are defined in the form [(s)moniker].[identifier], [(t)moniker].[(target)identifier]. The details of each connection end are stored in square brackets in the form '>'|'X","composition'|'aggregation'|'"','[lower]','[upper]. The first entry in this expression specifies whether the connection end is navigable (>) or not navigable (X), the second entry specifies

the kind of the connection end (i.e. composition, aggregation or neither (empty string) and the last two entries specify the lower and upper cardinalities of the connection end. Connection content (e.g. attributes and methods) is enclosed in curly brackets as previously shown.

Inheritance relationships are represented by the keyword Inheritance, followed by the name of the (inh)identifier, if defined. Then all subtype identifiers are listed in a comma-separated list followed by an arrow (->) and a comma-separated list of all (supertype)identifiers. Optionally, the inheritance characteristics (complete, incomplete, disjoint, overlapping) can be displayed in brackets as needed.

5.3 Textual User-defined Language

Like diagrammatic user-defined languages, textual user-defined languages are defined using visualizers. The textual visualizer definition metamodel is displayed in Figure 5.4. The meta metamodel mainly revolves around the concept of Values which are retrieved from the abstract syntax model and Literals which are static text (e.g. keywords) in the concrete syntax. A textual concrete syntax is defined by attaching a TextualVisualizer, inheriting from AbstractUserDefinedVisualizer, to a model element. The concrete syntax itself is defined by TextualVisualizationDescriptors which are contained in the TextualVisualizer.

Literal and Value specialize TextualVisualizationDescriptor and are used to define the textual concrete syntax. The literal attribute of Literal defines static, unchangeable text to display, e.g. keywords. Values can be retrieved in different ways from the abstract syntax model. AttributeValue displays the value of an attribute through data type sensitive editing features, e.g. for an attribute of an enumeration data type all possible enumeration values can be selected from a list. This context-sensitive AttributeValue is further configured by applying its subclasses EnumerationValue and BooleanValue which can be used to further configure the visualization of attributes of type boolean and enumerations. The BooleanValue type can be configured to show a literal representing the true value (trueLiteral) and a literal representing the false value (falseLiteral). Additionally, the concrete syntax can be configured to not show a visualization if the attribute is of value *false* (hideFalse = true). An example for this is the

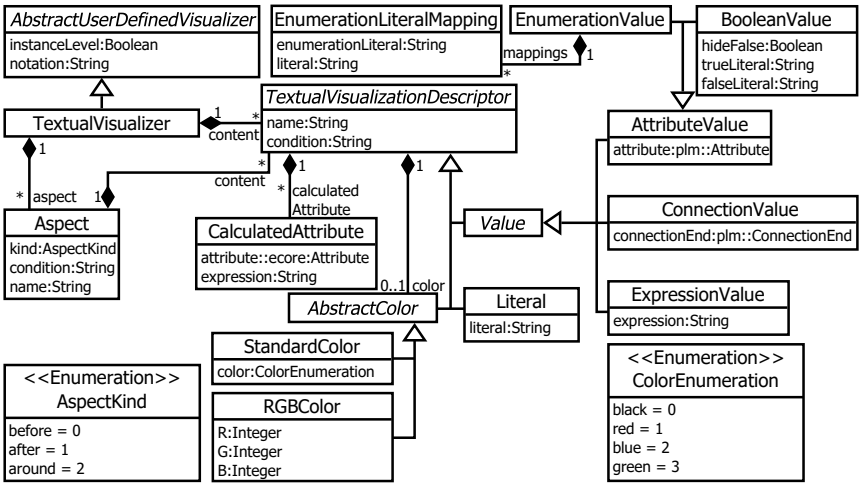


Figure 5.4: Textual visualizer metamodel.

Java syntax for declaring a class to be static or not. If the static keyword is present the static attribute of a class is set to true. If the keyword static is not visible the class' static attribute is set to false. Enumerations are visualized through the definition of an EnumerationValue which allows enumerationLiterals to be mapped to literals through EnumerationLiteralMappings. ConnectionValue displays the visualization of all model elements connected via the specified connectionEnd and ExpressionValue displays a non editable string calculated by a constraint language expression.

Values and Literals are highlighted using AbstractColor via the color reference of TextualVisualizationDescriptor. Two kinds of color are available: first, RGBColor supporting the definition of arbitrary colors through Red, Blue and Green values; and second, StandardColor supporting the enumeration literals black, red, blue and green of ColorEnumeration at the time of writing. To enable context-sensitive textual visualization, the CalculatedAttribute metaclass is referenced by TextualVisualizationDescriptor. The CalculatedAttribute sets the specified attribute to the result of the specified expression. Furthermore, it can be determined if a Value or Literal is visible by setting the condition attribute inherited from TextualVisualizationDescriptor.

Aspect-orientation of the user-defined visualization definition is supported

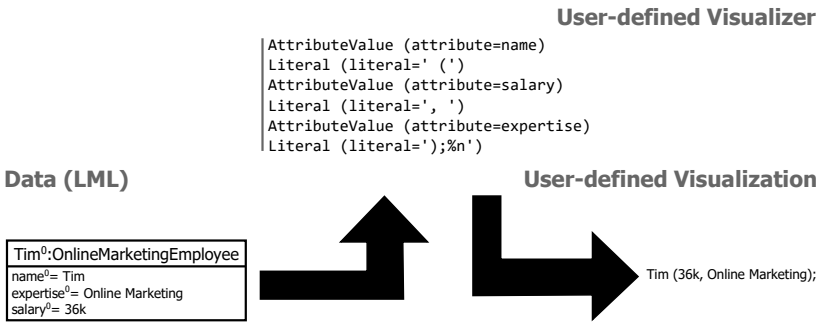


Figure 5.5: Tim visualized in the textual company structure modeling language.

through the Aspect metaclass. The name attribute defines which join point an Aspect is applied to. The condition attribute can be used to determine under what condition an Aspect is applied to a join point. An Aspect can be of kind before — placing the content of an Aspect before the join point, after — placing the content of an Aspect after the join point, or around — replacing the join point with the content of an Aspect. Literals and Values can be defined as join points to which an Aspect can contribute by setting their name attribute, inherited from TextualVisualizationDescriptor, to a unique value.

5.3.1 Textual Visualizer Metamodel Example

An example of the application of a user-defined textual visualizer is demonstrated in Figure 5.5. The top of the figure shows a textual user-defined visualizer for visualizing employees. The intention of the visualizer is to first show an employee's name, list the salary and expertise in brackets and terminate with a semicolon. For this purpose first an AttributeValue mapping pointing to the name attribute of the employee is created. This mapping is followed by the definition of a space and the start of the employee's details section (() using a Literal. In this section first the salary of the employee in question is displayed by an AttributeValue mapping pointing to the salary attribute of the employee. This mapping to the salary is separated by a comma and a space (a Literal defining ', ') and then the employee's expertise (an AttributeValue mapping pointing to expertise). The details section is then closed with a bracket and the employee is terminated with a semicolon followed by a line break defined through a Literal

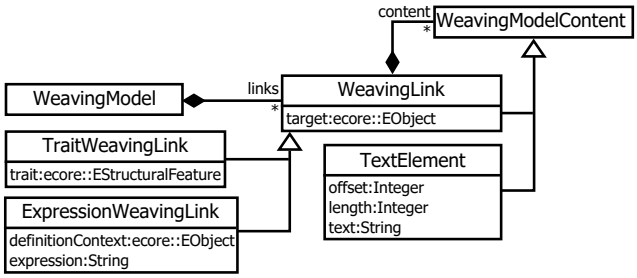


Figure 5.6: Textual weaving model.

();%n) in the textual user-defined visualizer.

The lower left of Figure 5.5 shows Tim, an employee to be textually visualized using the LML notation. The two arrows next to Tim indicate that the user-defined textual visualizer is applied to Tim resulting in the textual visualization in the figure's lower right. In this textual visualization Tim is visualized as an employee earning 36k per year with Online Marketing as his expertise.

5.4 Textual Weaving Model

To realize the projectional textual editor it is necessary to know which part of the text displayed to the modeler represents which part of the model in the underlying abstract syntax model representation. For this reason, a link between the edited text and the underlying model is established by employing an instance of the weaving metamodel displayed in Figure 5.6. This weaving model consists of four metaclasses specializing WeavingModelContent — TextElement and WeavingLink with its subclasses TraitWeavingLink and ExpressionWeavingLink. The target attribute of a WeavingLink points to a model element as a whole to visualize in the text. The WeavingLinks stored in the content of a WeavingLink then either point to traits to be visualized (TraitWeavingLink), store an expression used to calculate the displayed text (ExpressionWeavingLink) or point to contained and connected model elements (WeavingLink). The trait attribute of the TraitWeavingLink stores to which trait of a model element a weaving link maps to. In case of a weaving link for an ontological attribute this can for instance be the value of the attribute, the durability of the attribute or the name of

the attribute represented by text. ExpressionWeavingLinks store the expression used to calculate the textual visualization in their expression attribute and the model element on which the expression is defined (definitionContext). The text representing a weaving link is expressed as TextElements which are also stored in the content attribute of WeavingLinks. TextElements store the offset and length of the text they map to. Additionally, a copy of the represented text is stored in the text attribute.

A scenario in which it is necessary to nest WeavingLinks within one another is when the text to be visualized is composed out of several trait values from the abstract syntax model. An example of this occurs when visualizing several ontological attribute values in a user-defined language. In this case a WeavingLink is established to the clabject containing the attributes and for each attribute one TraitWeavingLink is added to the content of this WeavingLink pointing to the attribute and its trait (i.e. the *value* trait in case of a user-defined language) to display in the text.

Several operations on the weaving model are required by a projectional text-based model editor to realize projectional textual editing in a multi-format modeling environment. These operations are 1. calculate TextElement offsets 2. edit model element trait, 3. delete model element and 4. add model element.

5.4.1 Textual Weaving Model Operations

The calculateOffset(weavingModel) operation, displayed in Algorithm 5.1, operating on the weavingModel recalculates the offsets of all TextElements and is called after each change to the underlying data model and each change to the text displayed in the model editor, because these two operations change the offsets of model elements represented in the projected text. The input to the algorithm is the weavingModel on which the offsets of all TextElements have to be recalculated. First, the algorithm retrieves a treeIterator for the weavingModel (line 1). The treeIterator traverses the containment tree of the weavingModel following a depth first approach. Calling hasNext(treeIterator) on the tree iterator checks if there are more elements to traverse and the next(treeIterator) operation retrieves the next element in the containment tree of the weavingModel from the treeIterator. Then the global currentOffset counter is set to its initial value, 0, in line 2. The main work of the algorithm is done by lines 3 to 9.

```
   Data: weavingModel
   Result: Recalculate all offsets after a change
 1 treeIterator ← createTreeIterator(weavingModel);
 2 currentOffset ← 0;
 3 while hasNext(treeIterator) do
 4 |   current ← next(treeIterator);
 5 |   if isTextElement(current) then
 6 |   |   setOffset(current, currentOffset);
 7 |   |   setLength(current);
 8 |   |   currentOffset ← currentOffset + length(current)
 9 |   end
10 end
```

Algorithm 5.1: The calculateOffset(weavingModel) operation.

In this part of the algorithm, the current model element is checked to see if it is of type TextElement (line 5). If this is the case the setOffset(current, currentOffset) operation (line 6) sets the value of the current TextElement to the globally stored currentOffset, the length attribute of the current TextElement is recalculated (line 7) and the value of the current TextElement's length attribute is added to the currentOffset (line 8).

The findTextElement(weavingModel, offset, searchStrategy) operation is specified in Algorithm 5.2. The algorithm retrieves a TextElement from a given weavingModel for a given offset applying a given searchStrategy. The algorithm first searches all TextElements for the offset which is closest to the cursor offset (line 1 - line 9). Then the algorithm performs a depth first search on the weavingModel's containment tree for the TextElements which have the minimum distance to the cursor offset (line 11 - line 21).

Each model element (current) is checked to determine if it is a WeavingLink or a TextElement using the isTextElement(current) (line 13) operation. If the current model element is a TextElement, it is determined whether the offset of the edited text is: 1. at the beginning or within the TextElement (line 14) or 2. at the end of the TextElement (line 18). In both cases the TextElement is added to the result. If the edited offset is placed between two text elements in the edited file, both conditions are executed and thus more than one result is potentially retrieved

```
   Data: weavingModel, offset, searchStrategy
   Result: Finds a TextElement for a given offset.
 1 offsets ← ∅, results ← ∅;
 2 iterator ← createTreeIterator(weavingModel);
 3 while hasNext(iterator) do
 4 |   current ← next(iterator);
 5 |   if ¬isTextEelement(current) then continue;
 6 |   if offset - offset(current) ≥ 0 then
 7 |       offsets ← offsets ∪ (offset - offset(current));
 8 end
 9 minDistance ← min(offsets);
10 iterator ← createTreeIterator(weavingModel);
11 while hasNext(iterator) do
12 |   current ← next(iterator);
13 |   if ¬isTextEelement(current) then continue;
   |   // in model element or at beginning
14 |   if offset - offset(current) = minDistance then
15 |   |   results ← results ∪ current;
16 |   |   break;
17 |   end
   |   // the end of a model element, when between two
18 |   if minDistance = 0 then
19 |   |   if offset - (offset(current)+length(current)) = minDistance then
   |   |     results ← results ∪ current;
20 |   end
21 end
22 if size(results > 1) then
23 |   if strategy = ModelElementPreferred then getModelElement(results);
24 |   if strategy = TraitPreferred then getTrait(results);
25 else
26 |   first(results);
27 end
```

Algorithm 5.2: The findTextElement(weavingModel, offset, searchStrategy) operation.

from the weaving model.

To resolve this ambiguity the algorithm is configured with different strategies when executed. The algorithm can either return the first model element (line 23), the first trait (line 24) or the first element no matter what the type of the weaved target (line 26) from the results set is. In case a search strategy is configured to prefer a trait or model element and more than one trait or model element is in the results list the element residing at the first position of the list is returned.

Data: weavingModel, offset, file, add, length
Result: Change the model trait in the abstract syntax.

```
1 textElement ← findTextElement(weavingModel, offset, TraitPreferred);
2 weavingLink ← getWeavingLink(textElement);
3 if add then
4 |   textElement, setText(substring(file, offset, length(textElement)) +
  |    length)
5 else
6 |   setText(textElement, substring(file, offset, length(textElement)) -
  |    length)
7 end
8 recalculateOffset(weavingModel);
9 setAbstractSyntaxValue(weavingLink, textElement);
```

Algorithm 5.3: The editTrait(weavingModel, offset, file, add, length) operation.

The editTrait(weavingModel, offset, file, add, length) operation as described in Algorithm 5.3 is responsible for transferring changes from the projection in the text editor to the abstract syntax model. The algorithm first searches for the TextElement at the current offset (line 1) using the findTextElement(weavingModel, offset, TraitPreferred) operation and retrieves its corresponding WeavingLink using the getWeavingLink(textElement) operation (line 2). Then the text of the textElement is set to the new value depending whether a character is being added (line 4) or removed (line 6). Finally, the offsets for the changed weaving model are recalculated (line 8) using the recalculateOffset(weavingModel) operation and the value is written into the abstract syntax representation of the edited model (line 9) using the setAbstractSyntaxValue(weavingLink, textElement)

operation.

The add and remove model element operations on the abstract syntax model representation work in a similar way to the edit operation described in Algorithm 5.3. The difference is that the remove operation deletes the WeavingLink containing the TextElement and the add operation adds a new WeavingLink and its corresponding TextElements to the model. Afterward, the offsets in the WeavingModel have to be recalculated. In the predefined modeling mode a modeler can use the textual model editor to add or remove all linguistic model elements (i.e. clabject, model, attribute), whereas in the user-defined mode the modeler can only add or delete clabjects.

Operations for synchronizing changes from the abstract syntax model to the text editor also work in a similar way to Algorithm 5.3. The main difference is that the weavingLink is not retrieved using an offset but using the edited model element (i.e. trait for the editTrait() operation) to which the weavingLink points. Then, the corresponding TextElement and the text editor are updated instead of the abstract syntax as described in Algorithm 5.3. Since the algorithms transporting information from the abstract syntax to the text editor are very similar to the algorithms transporting information in the other direction they are not explained in more detail here.

5.4.2 Textual Weaving Model Example

An example of a weaving model which weaves text representing the company structure modeling language example to its abstract syntax representation is displayed in Figure 5.7. The lower left side of the picture shows the Data which underlies the edited text. A tree-based visualization is chosen to represent the model which is edited by the textual projection on the right-hand side of the figure. In the Text-based, user-defined representation a company is represented by its name followed by its departments in curly brackets. Departments are represented in the same way as companies but display their employees in curly brackets. Each employee is represented on a new line starting with its name, followed by the salary in brackets and terminated with a semicolon. The Weaving Model in the top center of the figure connects the data model in the bottom left with its textual representation in the bottom right.

For each clabject and attribute represented in the text, one WeavingLink is

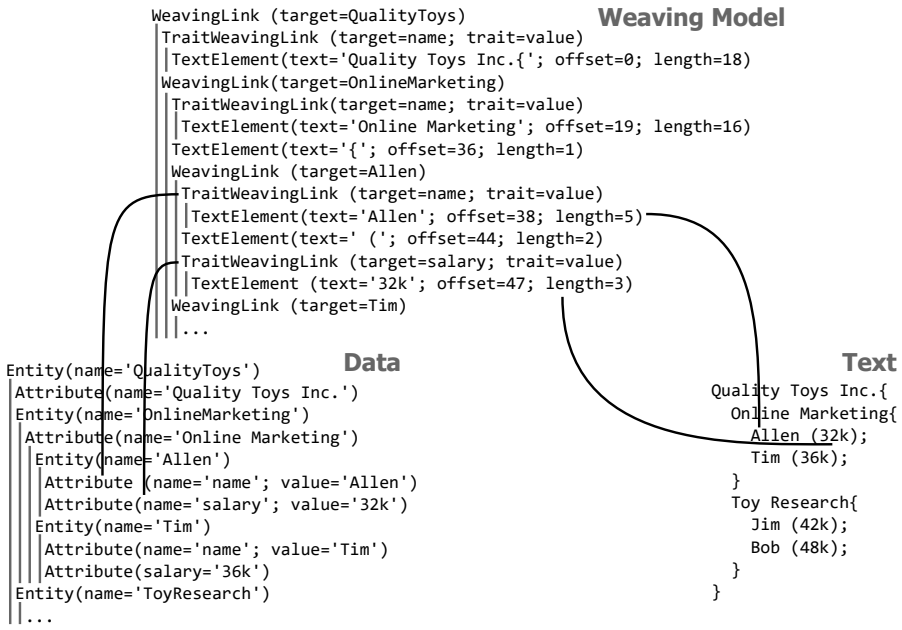


Figure 5.7: Textual weaving model for the company structure modeling language example.

created to the corresponding element in the abstract syntax representation of the model. The weaving model shows only a few WeavingLinks for space reasons. Where a WeavingLink points to is indicated by the target attribute value specified in the brackets of the WeavingLink's textual representation in the Weaving Model tree. Here, WeavingLinks are shown for the clabjects Quality Toys Inc., Online Marketing, Allen and Tim. Also, TraitWeavingLinks are shown for the value trait (trait=value) of the name (target=name) and salary (target=salary) attributes of Allen. Each WeavingLink contains TextElements as children indicating the text representing the WeavingLink. These TextElements indicate first the text which they represent, followed by the offset at which the element starts in the textual visualization and the length. Hence, this model establishes a link for each piece of text in the textual model to its representation in the abstract syntax model. Two of these links are shown in Figure 5.7 by solid black lines. These lines connect the name and salary of Allen with their representations in the abstract syntax representation and their textual visualization.

Chapter 6

The Tabular Format

The tabular visualization format organizes model content in rows and columns. The information belonging to a model element is contained in a row and is organized into columns based on the model element's properties. Compared to the two previously introduced formats it is clear that the tabular format is neither a spatial format like diagrams nor a sequential format like text. It therefore shares neither the advantages nor disadvantages of these formats. The tabular format is most suitable for showing huge amounts of data in a condensed way with sophisticated filtering, grouping, ordering and sorting mechanisms. This makes the format a perfect fit for data analysis and simulation tasks.

6.1 Tabular Predefined Language

When applying the tabular editing paradigm to linguistic modeling, the entities contained in a model element (e.g. clabjects contained in a level) are visualized as rows and their linguistic attributes (e.g. name, potency) and connections (e.g. list of an entity's ontological attributes connected via containment references) as columns. A problem of this mapping of abstract syntax elements to columns is that the content of a table can be of different linguistic types characterized by different linguistic traits. A level for example can contain amongst others inheritance relationships, connections and entities, all featuring a different set of traits. To combine all these model elements of

different linguistic types featuring different sets of traits in one single table, content sensitive filtering and displaying algorithms have to be employed on the table content.

To determine what is viewed in a table a *context* and *viewpoint* are set. The context is the model element whose content is to be displayed in a table. For a level, all clabjects, connections and inheritance relationships are shown. For a clabject all attributes, methods, contained clabjects, connections, and inheritance relationships are displayed. The viewpoint determines the columns and content displayed in a table. If for example the viewpoint is set to clabject only instances of the linguistic clabject type are displayed showing the linguistic traits of clabject.

```
Data: modelElements
Result: Find the smallest common linguistic type.
1  viewpoint ← plm::Element;
2  apply ← true;
3  while apply do
4      for subtype ∈ subTypes(viewpoint) do
5          apply ← true;
6          for modelElement ∈ modelElements do
7              if ¬conforms(modelElement, subtype) then
8                  apply ← false;
9                  break;
10             end
11         end
12         if applies then
13             viewpoint ← subtype;
14             break;
15         end
16     end
17     if subTypes(viewpoint) = ∅ then break;
18 end
19 return viewpoint;
```

Algorithm 6.1: Search for the most concrete common linguistic type.

When a table is opened the initial viewpoint is the most concrete common linguistic type of the content of the table's context. The algorithm determining the most concrete common linguistic type of a set of instances is shown in Algorithm 6.1. The input to the algorithm are all modelElements to be visualized in the table. The algorithm starts with the element at the root of the linguistic metamodel inheritance hierarchy, Element, as initial viewpoint (line 1). From this metaclass it traverses down the inheritance tree (line 4) until a viewpoint is found which has only subtypes to which not all modelElements conform (line 3). Conformance of modelElements to the subtype of the viewpoint currently under investigation is checked in line 6 and 7. If one modelElement does not conform to the currently investigated subtype, the apply variable is set to *false* indicating that currently no subtype suitable as viewpoint is found and the search is continued with the next subtype of the current viewpoint (lines 8 and 9). If a subtype is found to which all modelElements conform, it is used as the new viewpoint (line 13) and the subclasses of this viewpoint are searched for a most concrete common type (line 14) for the modelElements to be visualized. The algorithm terminates if the newly discovered viewpoint does not have any subtypes (lines 17). Alternatively, the algorithm also terminates when the first viewpoint is discovered which does not have any subtype to which all modelElements conform.

While working with the table the user can change this preselected viewpoint. So for example it can be switched from Clabject to Entity. After changing the filter to Entity the table shows Entities only but with all linguistic traits of Entity. Also the context of a table can be changed, by double clicking a row in the table. After changing the context of a table the viewpoint is reset to the most concrete common linguistic type of the new context's content.

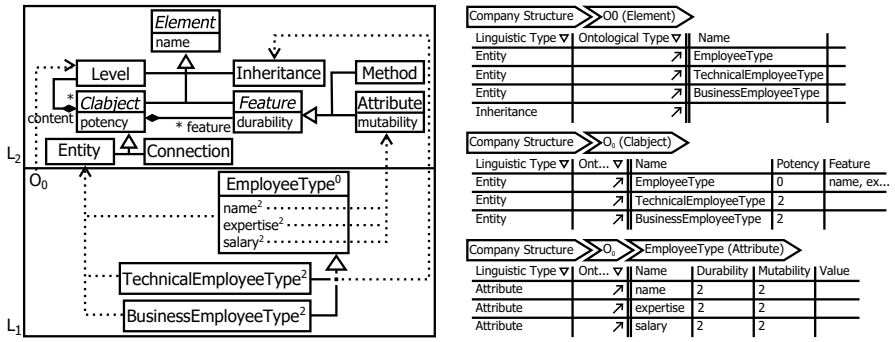
Displaying and filtering content based on a viewpoint and context is demonstrated in Figure 6.1 which shows the company structure modeling language modeled in the predefined diagrammatic LML notation on the left-hand side (Figure 6.1(a)) and three corresponding predefined tabular visualizations on the right-hand side (Firgue 6.1(b)). The first of the three tables shows the O_0-level set as context. The content of this table is equal to the content of the Company Structure model's L_1 level shown in Figure 6.1(a), which is limited to the O_0 level for space reasons. The current context of the table is indicated by

the horizontally aligned arrow signs above the table, also referred to as breadcrumbs [114]. Here *location breadcrumbs* are used to indicate the location at which the current context of the table is located within the edited deep model. Each row in the table corresponds to one model element residing in level O_0. The initial viewpoint is the most concrete common type of all O_0 content which is Element as indicated by the brackets in the breadcrumb. The first column shows the linguistic type of the model element represented by the row and the second column shows the ontological type of the model element. Here, model elements of the linguistic type Entity and Inheritance are displayed in the table. Since O_0 is the highest level, no ontological types are displayed for the O_0 content. The rectangle pointing down next to the column heading of Linguistic Type and Ontological Type indicate that the column values can be used to filter the viewed model elements. The list available for filtering elements contains all the values present in a column. In case of the Linguistic Type column Entity and Inheritance can be selected. Additional columns can be included in the table but are no shown in Figure 6.1(a) for space reasons. These include a possible third and fourth column displaying the supertypes and subtypes of the model elements and a possible fifth column showing model elements connected to the model element via connections. All these cells can display multiple values since multiple inheritance, multiple connections and multiple linguistic/ontological classification relationships are supported. Clicking on the arrow button in one of these columns opens a table displaying the cell content in its own table.

The following columns separated by a vertically double lined border show the linguistic trait values of the model elements. The displayed columns depend on the viewpoint which is the most concrete common type of the context's content, — Element, in this example. Hence, the linguistic name trait is displayed for editing because this is the only linguistic trait of Element.

The table in the middle of Figure 6.1(b) shows the content of the O_0 level with the viewpoint set to the linguistic type *Clabject* as indicated by the last segment of the middle table's breadcrumb (O_0 (Clabject)). Three Clabjects, EmployeeType, TechnicalEmployeeType and BusinessEmployeeType conforming to the linguistic type Clabject are displayed. In addition to the name trait the traits potency and feature common to all Clabjects are included in the table.

The bottom table displays the effect of setting the context to EmployeeType



(a) (b)

Figure 6.1: EmployeeType LML model (a) and its predefined tabular representation (b).

by double clicking it in the middle table, as indicated by the table's breadcrumb. The EmployeeType clabject contains three Attributes and no operations. For these attributes the name, durability, mutability and value traits are displayed.

New model elements can be instantiated by adding a new line to a table and selecting a linguistic or ontological type from the Linguistic Type or Ontological Type column. If a linguistic type is selected, the ontological type is left blank by default. If an ontological type is selected, the linguistic type of the ontological type is automatically selected. The ontological types available for selection are context-sensitive to the current location of the model element edited by the table. Delete operations can also be invoked on any row to delete the corresponding model elements.

6.2 Tabular User-defined Language

Like diagrammatic and textual user-defined languages, tabular user-defined languages are defined using visualizers tailored towards the definition of tabular languages. The visualization is then applied to each model element by invoking the corresponding format-aware visualization search algorithm. In contrast to the previously described formats at least two visualizers always participate in the visualization of one model element. One is the visualizer configuring the table container that is the currently selected model element in

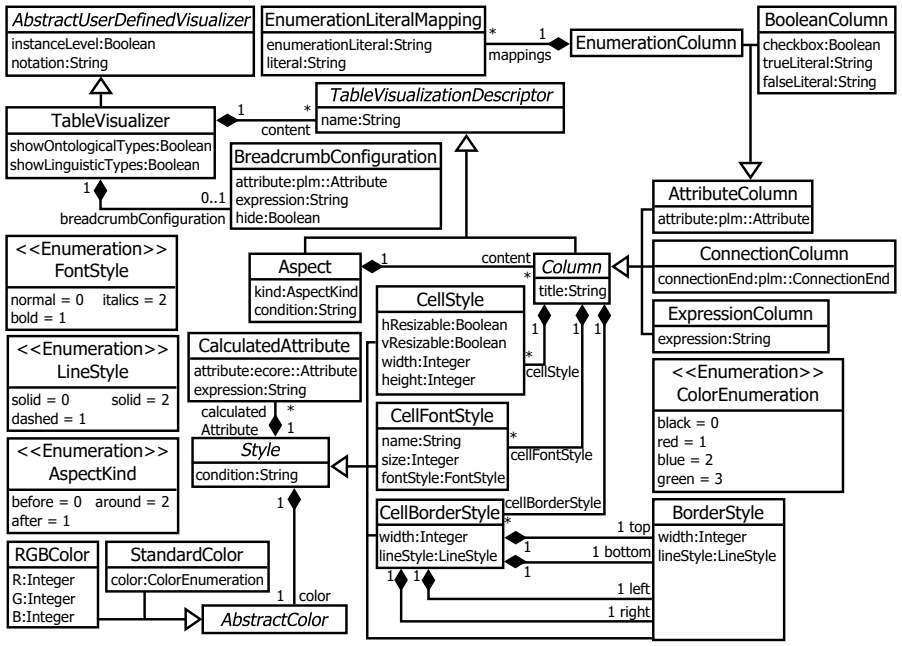


Figure 6.2: Tabular visualizer metamodel.

the editor (i.e. the context). This visualizer is responsible for global settings such as the breadcrumb and the ontological and linguistic type columns. The other visualizers, defined on the table's content, deal with the content of the rendered table (model elements owned by the selected model element). The actual visualizer content, e.g. mappings of attributes to columns, is used for their rendering.

The metamodel to define user-defined tabular languages is shown in Figure 6.2. TableVisualizers own two attributes, showOntologicalTypes and showLinguisticTypes which determine whether the columns for ontological and linguistic types are shown. The breadcrumb is configured by adding a BreadcrumbConfiguration to the visualizer. The breadcrumb can be set to be hidden (hide attribute) and to either display an attribute value or the result of an expression defined in a deep constraint language. If the attribute and expression are not set, the linguistic name trait is displayed in the breadcrumb. The details of the table visualized by the TableVisualizer are described by its owned TableVisualizationDescriptors. These describe the Columns found in the tabular

user-defined language. Columns can be either mapped to ontological attributes (AttributeColumn), connections (ConnectionColumn) or can display values calculated by a constraint expression (ExpressionColumn). The title attribute of Column sets the title that is displayed for the column. If this attribute is not set for AttributeColumn and ConnectionColumn, the mapped attribute name or connection end name is used as the column title. If the title attribute is not set for ExpressionColumn the title of the visualized column is blank.

The attribute which is displayed in an AttributeColumn is specified by its attribute attribute. This column offers data type specific cell editors, e.g. for an attribute of data type boolean a cell displaying a checkbox is displayed. The AttributeColumn can be further refined for enumeration and boolean attributes by applying the corresponding EnumerationColumn and BooleanColumn subtypes. BooleanColumns can be configured to display a checkbox indicating true/false values (checkbox attribute set to true) or to display a drop-down list for true/false value selection (checkbox attribute set to false). This drop-down list offers *true* and *false* as default values. The literals displayed in the drop-down list can be customized by setting values for the trueLiteral and falseLiteral attributes. In case that the title attribute is not set, the trueLiteral is displayed as the column title of a BooleanColumn if specified, otherwise the attribute name is displayed. EnumerationColumns by default show the names of the literals defined in the enumerations in a drop-down list. These values can be refined to more human friendly text by defining EnumerationLiteralMappings for each enumeration literal. The enumerationLiteral attribute maps the corresponding enumerationLiteral of the underlying enumeration to the text displayed by the UI as defined in the literal attribute. The ConnectionColumn is used to display all Clabjects which are connected with one clabject via the specified connectionEnd. This mapping is used in tabular languages to enable navigation from one element to another. Furthermore, it can be used to allow convenient creation of connections between clabjects. The tabular user interface supports navigation over connections in the way previously described for the ontological types column. ExpressionColumns display the value defined by the expression attribute. Arbitrary expressions defined in a deep constraint language can be used to determine the value displayed in this column. This prohibits the content of this type of column from being edited because the value placed in the column

cannot be related to one single attribute.

The visualization of Columns is configured via CellStyle, FontStyle and BorderStyle instances. CellStyle configures the visualization of the cell which includes the width and height of the cell as well as the property of being horizontally resizable (hResizable) and vertically resizable (vResizable). The background color of the cell is set by the color reference inherited from Style. There are two ways of specifying colors in tabular user-defined languages. The first is using one of the predefined colors which, at the time of writing, are black, red, blue and green. The second is using R, G, B value-based color defined through RGBColor. To configure the visualization of cell borders in a column CellBorderStyles are used. For each cell the top, bottom, left and right are configured separately if required. Each border of the cell can have a color, width and lineStyle configured. If the same values shall be applied to more than one border these can also be set in the CellBorderStyle and then refined using top, bottom, left and right BorderStyles. The font in a cell is controlled by specifying a CellFontStyle, which supports the definition of a fontName, size and fontStyle (i.e. normal, italics and bold).

The described styles are applied to all cells in a column by default. This default behavior can be modified to apply a style to certain cells only by setting the condition attribute of the Style metaclass to an expression defined in a deep constraint language. Using this feature, multiple styles can be defined for one column and applied in a context-sensitive way determined by the table's underlying data. Also the values defined for the Style's attributes can be set in a context-sensitive way. To do so one or more CalculatedAttributes have to be added to a Style. The attribute attribute defines for which Style attribute (e.g. width, lineStyle) the value shall be calculated. The expression attribute contains a deep constraint expression to calculate the value of the specified attribute.

As in the previously presented diagrammatic and textual user-defined language definition metamodels, aspect orientation is also supported in the tabular user-defined language definition model. Columns can be marked as join points by setting their name attribute inherited from TableVisualizationDescriptor to a unique value. Aspects can then be defined to provide content to the join point and an application strategy (before, after, around) for applying aspects to the join point through the kind attribute.

In the user-defined table editor the concepts of viewpoint and context are

available as in the predefined table editor. Equally, the context is the container of the model elements to be displayed in the table editor. The viewpoint, however, is moved from the linguistic dimension, declaring linguistic types, to the ontological dimension, declaring ontological types. The types available for viewpoint definition are all types of clabjects and connections contained by the context and located in the context classifying level.

In the tabular, user-defined visualization language the content of the currently selected model element is always displayed in a table. Thus, model elements of different ontological types are often displayed within the same table. This raises a similar problem to that previously described in the context of predefined tabular languages. Model elements viewed in the same table are described by different sets of ontological features. To manage this situation the most concrete common type search algorithm is applied again but on the ontological dimension searching the most concrete common ontological type instead of the linguistic dimension. The displayed columns, however, are not derived from the ontological attributes and connections defined on the most concrete ontological type but through the visualizers attached to the ontological types. By combining the visualizers of the most concrete common ontological type and its subtypes compatibility between the different visualizers is ensured. All ontological types of model elements contained in the context are available as viewpoint candidates, as are their supertypes.

When initially opening a user-defined table editor, the viewpoint is set to the most concrete common ontological type of the context's content (i.e. table editor content). Furthermore, only entities are considered as candidates for the initial viewpoint. A modeler, however, can set the viewpoint to a connection at any time. An ontological model does not necessarily have a single root element in the inheritance hierarchy like the linguistic model of the deep modeling approach presented in this thesis. Thus, a bottom-up search for the most concrete common supertype is performed in contrast to the top-down search for the most concrete common linguistic type search applied to linguistic classifiers. The bottom-up search starts at the most concrete types of the table content, progressively traversing the inheritance trees towards more abstract types. The ontological viewpoint search algorithm is displayed in Algorithm 6.2.

First, all ontological types of the modelEntities to be displayed are collected in a queue (line 2). These are then iterated over (line 3 + 4). Each type is checked to determine whether it is an ontological type of all the entities to be visualized (lines 5 - 10). If a type for all modelEntities is found the search is stopped (lines 11 - 14) and the most concrete common ontological type for all modelEntities is found. Otherwise, the direct supertypes of the current type under investigation are appended to the typesQueue and the search is continued (line 15). If no common viewpoint is found, no viewpoint is returned resulting in an empty table to be initially rendered. In this case the *viewpoint* is selected by the user.

Data: modelEntities
Result: Find the smallest common ontological type.

```
1  viewpoint ← ∅;
2  typesQueue ← getAllOntologicalTypes(modelEntities);
3  while hasNext(typesQueue) do
4      type ← poll(typesQueue);
5      applicable ← true;
6      for modelEntity ∈ modelEntities do
7          if ¬ conformsTo(modelEntity, type) then
8              applicable ← false;
9          end
10     end
11     if applicable then
12         viewpoint ← type;
13         break;
14     end
15     append(typesQueue, getDirectSupertypes(type))
16 end
17 return viewpoint;
```

Algorithm 6.2: Search for the most concrete common ontological type.

In contrast to the predefined table editor which aims to display a minimal number of columns, the user-defined table editor aims to display as many columns as possible. This, for example, prohibits tables with a low number

of columns from arising when one single ontological type has a rather small visualizer defined in contrast to the others. The columns displayed in a user-defined table are the union of all the columns defined in the visualizers of all table content determined through the combination of context and viewpoint. In addition, the combined visualizers have to be defined in the notation intended for visualization. Algorithm 6.3 shows the visualizer column merge algorithm.

```
   Data: modelElements, notation
   Result: Merge the visualizers.
1  mergedVisualizer;
2  for modelElement ∈ modelElements do
3  |  visualizer ← searchVisualizer(modelElement, notation, true);
4  |  columns ← getColumns(visualizer);
5  |  for column ∈ columns do
6  |  |  if ¬ column ∈ getColumns(mergedVisualizer) then
   |  |  |  appendColumn(mergedVisualizer, column);
7  |  end
8  end
9  return mergedVisualizer;
```

Algorithm 6.3: Merge the visualizers columns.

The algorithm iterates through all modelElements to be visualized (line 2), searches their visualizer through the visualizer search algorithm (line 3) for the notation to be visualized and merges it in case aspects are collected by the search algorithm (true). Then the columns of the visualizer are retrieved (line 4), iterated over and appended to the mergedVisualizer in case they are not already existing (line 5 - 7).

6.2.1 Tabular Visualizer Metamodel Example

The usage of the tabular user-defined language definition metamodel is demonstrated in Figure 6.3. The data to be visualized in the user-defined tabular visualization is shown at the bottom left of the figure — the Toy Research department containing two research employees, Jim and Bob. When a model

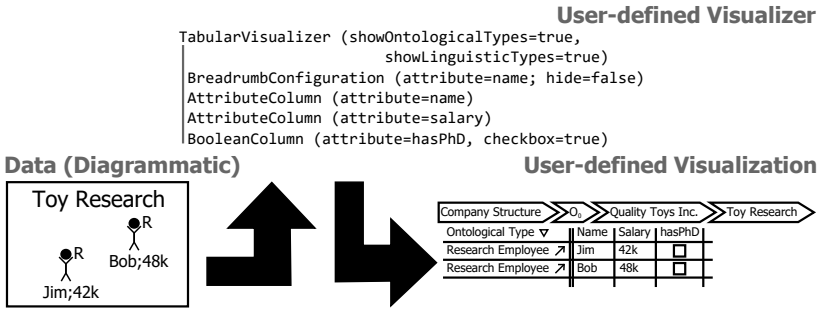


Figure 6.3: The Toy Research department content visualized in the tabular company structure modeling language.

element is selected in the tabular visualization its content is visualized. Here, the two employees Jim and Bob are visualized when selecting the Toy Research compartment in the tabular user-defined language.

The visualization definition of the two research employees Jim and Bob is shown in the top center of Figure 6.3. The context of the table displayed in the lower right of the figure, however, is the Toy Research department. The visualizer of the Toy Research department (not shown in Figure 6.3) is configured to hide the linguistic type column (showLinguisticType=false) but show the ontological type column (showOntologicalType=true) and the breadcrumb is configured to be visible (hide=false). Furthermore, the name attribute of the department is displayed in the breadcrumb (attribute=name). These configurations, showLinguisticType, showOntologicalType, hide and the string displayed in the breadcrumb are taken from the context (Toy Research) of the visualized table and not the content (here Jim and Bob) because the different contents of a table can have contradictory breadcrumb configurations.

Three columns are displayed for Jim and Bob. The first two are AttributeColumns displaying the name attribute (attribute=name) and the salary attribute (attribute=salary). The third is a BooleanColumn displaying whether the research employee has a PhD (attribute=hasPhD). This is indicated through a checkbox and not a selection of two literals showing either true or false (checkbox=true). The hasPhD attribute of Research Employees was not defined in previous versions of the company structure modeling language running example. It has been explicitly added to this example to demonstrate the usage of the BooleanColumn

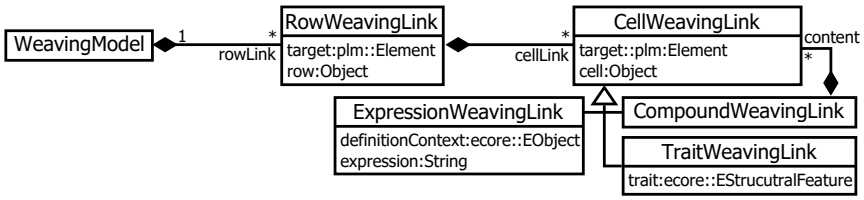


Figure 6.4: Tabular weaving model.

meta type.

The bottom right side of Figure 6.3 shows the result of applying the tabular user-defined visualizer to the content of Toy Research, which is the current context indicated by the breadcrumb above the table. A table featuring four columns and two rows is created from the input. The first column displays the Ontological Type of the model element represented in the rows, the second represents the name, the third represents the salary and the fourth represents the PhD ownership. The two research employees Jim (first row) and Bob (second row) and their corresponding attribute values are displayed as content of the table. To add a new research employee, the ontological type Research Employee has to be selected in the Ontological Type column of the last empty row in the table.

6.3 Tabular Weaving Model

The tabular weaving model which is responsible for connecting cells in a table to the underlying abstract syntax representation is shown in Figure 6.4. This model consists of a WeavingModel containing RowWeavingLinks which connect rows to their corresponding target elements in the abstract syntax model. The RowWeavingLinks contain CellWeavingLinks linking cells to target model elements. CellWeavingLink is specialized by TraitWeavingLink, ExpressionWeavingLink and CompoundWeavingLink. TraitWeavingLinks map cells to traits, ExpressionWeavingLinks map cells to expressions and CompoundWeavingLinks map representations of multiple model elements into one cell through their contained WeavingLinks. In the predefined modeling language the target of a RowWeavingLink and its TraitWeavingLinks is identical because the cells edit traits of the model

element displayed in a row, e.g. the potency trait of a clabject. In the user-defined modeling language, however, the cells display the trait value of ontological attributes of the model elements displayed in the rows (e.g. the value of the salary attribute of an employee). Users can then edit the values of the ontological attributes of a model element displayed in a row. Hence, the target attribute of RowWeavingLink points to the model element edited in a row, the target of a TraitWeavingLink points to the ontological attribute edited by a cell and the TraitWeavingLink's trait points to the value trait of the ontological attribute.

In addition to the information stored in the weaving model presented here the layout could also be stored in a tabular weaving model as in the diagrammatic and textual formats. Such layout information would include the order of rows, the order of cells and their width and height. The tabular format, however, is a format which does not rely heavily on user-defined layouts since modelers typically use the automatic layout features of table editors such as filtering, automatic ordering and automatic width and height adjustments of columns and rows. Hence, the current version of the tabular weaving model does not store layout information.

As with the diagrammatic weaving model, the manipulation operations — add, remove, move and edit — are not explained in detail here. In contrast to the textual weaving model, operations on one part of the tabular weaving model do not influence other parts of the model because layout is not considered in the tabular weaving model. Furthermore, both representations of the model rely on pointers in memory, making it trivial to trace a model element from one side of the weaving model to the other. In general, the tabular weaving model works in the same way as the diagrammatic weaving model when it comes to weaving model manipulation operations.

6.3.1 Tabular Weaving Model Example

An example of how a tabular weaving model connects a tabular user-defined language to its underlying model represented in the abstract syntax is shown in Figure 6.5. The bottom left of the figure shows the Data model underlying the Table in a tree-based visualization. The same model is represented using a user-defined tabular language at the bottom right of the figure. The Weaving

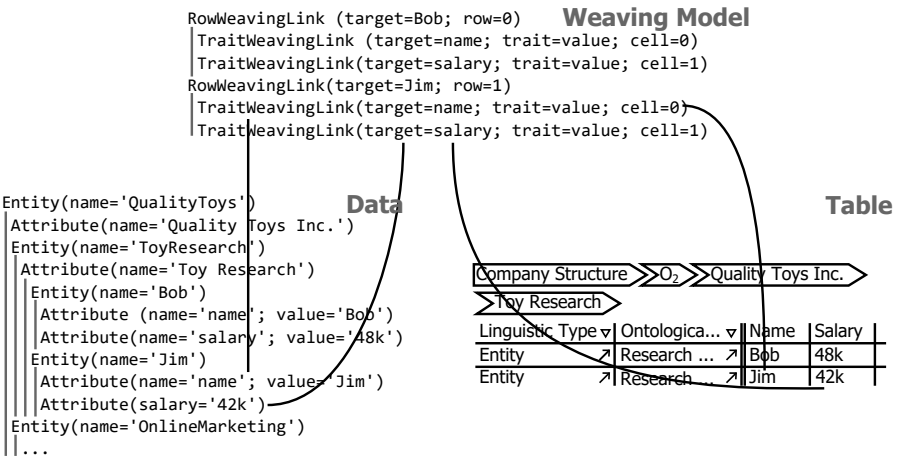


Figure 6.5: Tabular weaving model for the company structure modeling language example.

Model between the abstract syntax representation and its tabular visualization is displayed as a tree structure at the top of Figure 6.5. The breadcrumb of the tabular view shows that the context is the Toy Research department. Two rows are displayed representing the two employees Bob and Jim. The first two columns offer the linguistic and ontological types for navigation — Entity and Research Employee respectively. The next two columns display mappings to the name and salary attributes of Bob and Jim. The weaving model, shown at the top center of the figure, consists of two RowWeavingLinks, one for Bob and one for Jim. These RowWeavingLinks connect the table rows with the abstract syntax model representation of the model elements in memory. This, however, is hard to display in a tree view. Hence, the names of model elements and traits are used to represent the pointers of a weaving link's target and trait attributes to their corresponding abstract syntax model representations. The row and cell indexes are used to point to the table rows and cells representing the model elements and their traits. The columns of each row are mapped to the abstract syntax representation by TraitWeavingLinks, mapping cells to the value trait of ontological attributes in the abstract syntax model representation. For illustration purposes the pointers from the weaving model to the table and abstract syntax are represented by black solid lines for the ontological name and salary attribute of Jim.

Chapter 7

The Form-based Format

The goal of the form-based format is to provide a user interface to a model which looks and feels like a desktop or web application. A form displays the properties and relationships of one single model element. The forms for editing a model use the well known concepts of buttons, text boxes, checkboxes, radio buttons etc. from UI tool kits such as the standard widget toolkit (SWT) [172] or GTK+ [143]. Using form-based languages for editing models, user interfaces which hide all the complexity of a modeling environment can be created. Hence, this format is suitable when targeting audiences with few if any technical skill. The advantage of simplicity comes at the cost of power. On the one hand, when compared to the previously presented formats, form-based languages are not as efficient as text for entering huge amounts of data, not as effective as diagrams for displaying relationships between model entities and not as suitable as tables for working with huge amounts of data. On the other hand, form-based languages do not have some of the drawbacks of these formats such as the need to learn a technical, textual language, or the risk of errors in communication through bad diagram layout.

7.1 Form-based Predefined Language

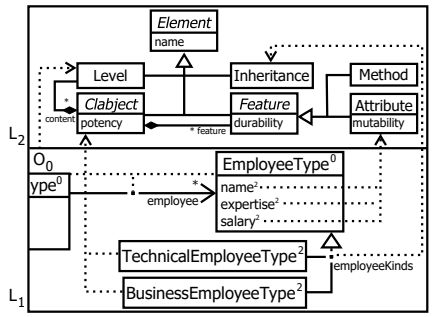
Example screens of the predefined form-based language covering all linguistic types of the applied deep modeling approach are shown in Figure 7.1. The

edited model is displayed in Figure 7.1(a) using the diagrammatic predefined LML representation of an excerpt of the company structure modeling language. In Figure 7.1(b) - (f), model elements are being edited using the predefined form-based format. The EmployeeType entity is being edited in Figure 7.1(b), the connection pointing to EmployeeType from its left-hand side in the diagrammatic format is being edited in Figure 7.1(c), the inheritance relationship specifying the subtypes of EmployeeType is being edited in Figure 7.1(d), the deep model containing all levels is being edited in Figure 7.1(e) and level O_0 is being edited in Figure 7.1(f). These figures exemplify the five kinds of forms available for editing a deep model using the predefined form-based language.

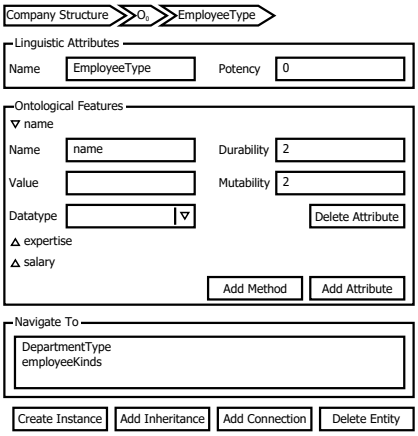
All forms of the predefined form-based format have a common structure. The top shows a breadcrumb indicating the location of the model element currently being edited and has additional navigation functions. The breadcrumb works in the same way as in the tabular format. Below the breadcrumbs the Linguistic Attributes, such as name and potency, are displayed for editing. The bottom of the dialog shows the Navigate To section which offers options to navigate the model, followed by buttons offering general operations on the currently edited model element such as deletion.

Between the Linguistic Attributes section and the Navigate To section, linguistic type specific sections are offered. For clabjects this is the Ontological Features section as shown for the EmployeeType entity in Figure 7.1(b) and the connection to EmployeeType in Figure 7.1(c). The single ontological features, i.e. attributes and methods, contained by a clabject are displayed using expandable sections in the Ontological Features section. In the case of EmployeeType (Figure 7.1(b)) the Ontological Features section displays the three ontological attributes of EmployeeType — name, expertise and salary. The only ontological attribute for which details are displayed in the example in Figure 7.1(b) is the name attribute. All other ontological features, expertise and salary, are collapsed. In each feature section a delete button is available to delete the feature, here the Delete Attribute button to delete the name feature of EmployeeType. At the bottom of the Ontological Properties section, the Add Method and Add Attribute buttons are offered to create new methods and attributes.

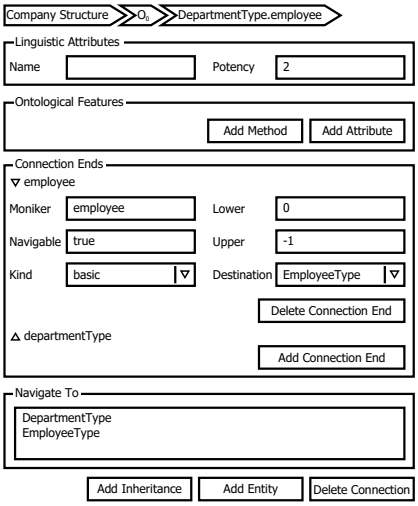
In the case of connections, an additional Connection Ends section is placed below the Ontological Features section as shown for the connection pointing to



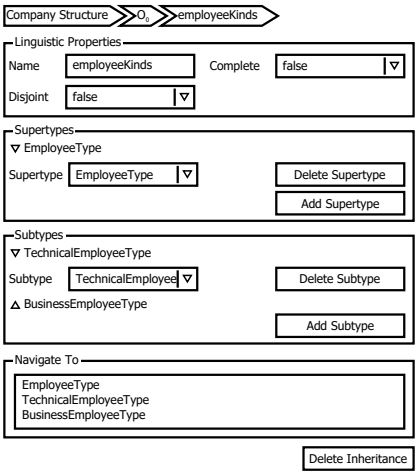
(a)



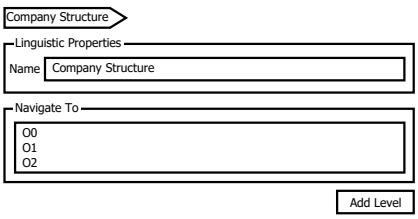
(b)



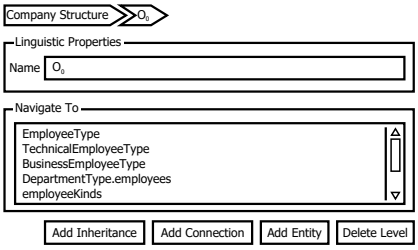
(c)



(d)



(e)



(f)

Figure 7.1: Company structure LML model (a) and its predefined form-based visualization of EmployeeType (b), DepartmentType.employee connection (c), employeeKinds inheritance (d), CompanyStructure deep model (e) and O_0 level (f).

EmployeeType which is being edited using the form-based predefined language in Figure 7.1(c). This Connection Ends section displays the connection's employee and departmentType connection ends. The employee connection end is expanded to show its details. Again, as in the case of ontological features, the text boxes can be used to set the linguistic attributes of the connection end and the Delete Connection End button can be used to delete the edited connection end. New connection ends are created via the Add Connection End button at the bottom of the Connection Ends section.

Forms displaying inheritance relationships offer the option to add or remove their super and subtype ends via the Supertypes and Subtypes sections. In Figure 7.1(d) the Supertypes section of the employeeKinds inheritance contains the inheritance relationship's supertype pointing to EmployeeType. The Subtypes sections displays two subtypes, one pointing to TechnicalEmployeeType and one to BusinessEmployeeType. The super and subtypes can be changed through the combo boxes displaying the connected model element and deleted through the *delete buttons* displayed next to them. The *add buttons* located at the bottom right of the corresponding sections are used to add new super or subtypes.

The form for editing the deep model which is the container of the levels is shown in Figure 7.1(e). It offers the Linguistic Attributes section and the Add Level button to edit the deep model. A Level is edited using the form displayed in Figure 7.1(f). Again the Linguistic Attributes section is offered to edit the name of the level, and Add Entity / Add Connection / Add Inheritance buttons are offered to add new entities, connections and inheritance relationships respectively.

A form with the newly created model element is shown after clicking one of the add buttons (e.g. the Add Level button of a deep model form or the Add Connection button of an entity form). When clicking one of the delete buttons (e.g. Delete Entity, Delete Connection) the model element's container is displayed after the delete operation has been completed.

7.2 Form-based User-defined Language

The visualizer metamodel for the definition of user-defined form-based languages is shown in Figure 7.2. Like the visualizers for all other formats the FormVisualizer inherits from AbstractUserDefinedVisualizer so that it can be at-

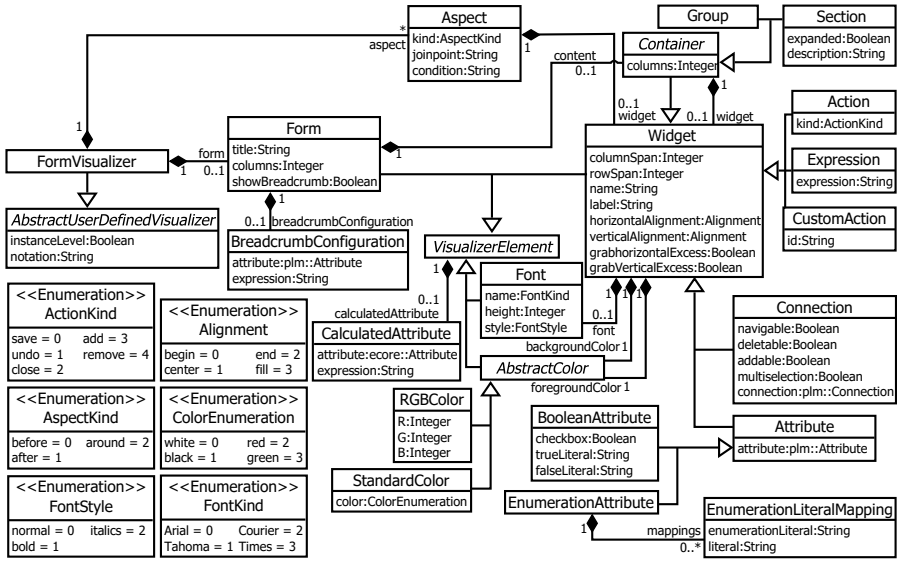


Figure 7.2: Form-based visualizer metamodel.

tached to instances of all linguistic types. All content in a FormVisualizer is a subclass of VisualizerElement which can store CalculatedAttributes. These CalculatedAttributes are used to set visualizer attributes based on the outcome of statements in a constraint expression. For instance, the title of a Form can be set to the name of the currently displayed ontological model element.

The FormVisualizer contains the Form describing the outer-most container of a statement in the form-based language. The Form can be configured by setting a title and the number of columns in its grid layout that arranges its content. For simplicity, the grid layout is used as the default layout in the form-based visualization implementation. The Form is further described by Containers which are either Groups or Sections which contain Containers or other Widgets for displaying and manipulating information using the form-based user-defined language. A form can be configured to show or hide a breadcrumb (showBreadcrumb). If a language engineer decides to show a breadcrumb in the form-based language, it can be further configured through a BreadcrumbConfiguration to either display the value of an ontological attribute or to show the value of an expression written in a deep constraint language.

Groups are containers surrounding their content with a border displaying

a label, inherited from Widget, in the top left corner of the border. This label serves as a caption summarizing the nature of a Group's content. A Section, however, does not surround its content with a border. It just displays a header at the section start. In contrast to a Group, a Section can be collapsed and expanded by the modeler. The initial state is configured by setting its expanded attribute to show or hide the section's content. Additionally, a description of the content located in the section can be displayed below the Section's label. Both, Section and Group can have the amount of columns they use to arrange content in a grid set via the columns attribute inherited from Container.

Containers contain Widgets which themselves can be Containers organizing form content or Widgets for viewing and manipulating model data. To execute actions on a model the Action metaclass is available providing default actions which are selected through the kind enumeration attribute of type ActionKind. These ActionKinds include amongst other things the save, undo and close operations. User-defined actions can be used in a form-based language by defining CustomActions. For these, a language engineer has to provide a label to be displayed for an action and an id identifying the operation to be performed when it is selected. In the form-based language actions are visualized as buttons.

To display and manipulate data from the model represented by the form-based language, Attribute, Connection and Expression are available. Attribute maps an ontological attribute to a text box. By default the name of the mapped attribute is displayed on the left of the text box. This text can be configured by setting the Attribute's label inherited from Widget. The visualized Attribute controls are data type sensitive like in the textual and tabular formats. The BooleanAttribute and EnumerationAttribute subtypes of Attribute offer advanced configuration options mapping attributes of enumeration or boolean data type. User-defined strings for displaying BooleanAttributes can be mapped to the true (trueLiteral) and false (falseLiteral) values to display more informative text than merely *true* or *false* to the user. To display a checkbox instead of a value selection the checkbox attribute is set to *true*. EnumerationAttributes map the single enumerationLiterals of an enumeration type to strings which are displayed to the modeler instead of the enumeration literals as defined in the modeling language's abstract syntax when editing attributes of an enumeration data type.

To display values which do not depend on one single ontological attribute but on a combination of several ontological attributes, an Expression in a deep constraint language can be defined to calculate the value to be displayed. Since they are calculated the results of Expressions are not editable by the user. The label attribute defines the text shown next to the calculated expression value. The label attribute of an Expression must be explicitly set because an Expression cannot be mapped to one single attribute from which the title for the calculated value can be derived.

Connection displays all entities connected via the specified connection. It can be configured to allow: 1. navigation via the connection (navigable), 2. the deletion of the connection and the connected elements (deletable), 3. the addition of new elements by creating a new connection (addable), 4. the selection of multiple connections at the same time and the execution of operations such as delete on the selected items (multiselection) or 5. the overriding of the default label's text displayed next to the list of all connected elements (label inherited from widget).

The font, backgroundColor and foregroundColor of Widgets are set using the respective attributes. The used Font is specified by defining the name of the font, size and style (i.e. normal, italics and bold). The foregroundColor and backgroundColor are specified either through a StandardColor or an RGBColor. A StandardColor offers the default colors defined in the ColorEnumeration. Even though this enumeration is limit to three colors in Figure 7.2, it can be extended to as many colors as needed. The RGBColor can specify any color using Red, Green, and Blue values. Furthermore, all Widgets can be configured to span more than one row (rowSpan) and more than one column (columnSpan) in the grid of its container. Widgets can also be configured to occupy all vertical space (grabVerticalExcess) and occupy all horizontal space (grabHorizontalExcess). The horizontalAlignment and verticalAlignment attributes are used to determine the alignment of a Widget within a grid cell.

Aspect-orientation is supported by declaring Widgets as join points by setting their name attribute. Aspects can then provide Widgets to these join points through their content attribute. As with all other formats, Aspects can be configured via their kind attribute to replace the join point (around), place the widgets before the join point (before) or after the join point (after). The name

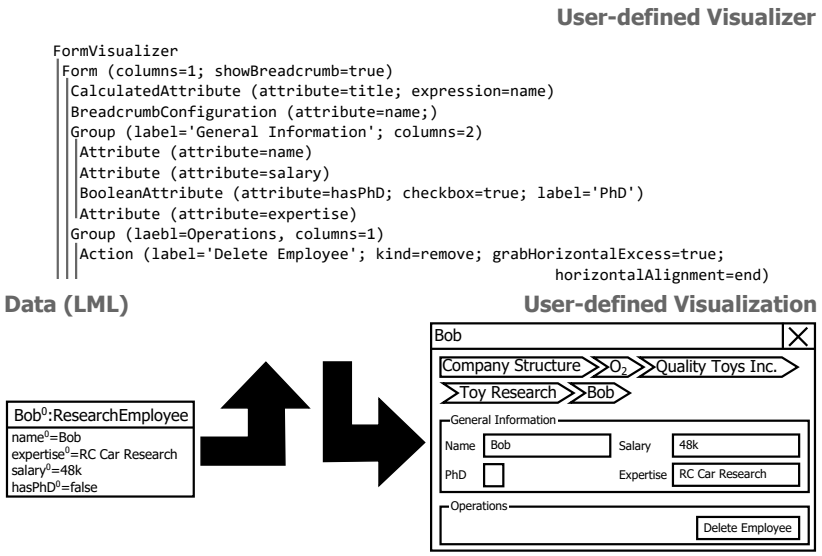


Figure 7.3: Bob visualized in the form-based company structure modeling language.

attribute identifies the join point to which the aspect contributes and the condition attribute specifies a condition defined in a deep constraint language which must hold true in order to apply the aspect.

7.2.1 Form-based Visualizer Metamodel Example

An example of the usage of the form-based user-defined visualization definition metamodel is shown in Figure 7.3. The bottom left of the figure displays Bob, an instance of ResearchEmployee, in the predefined diagrammatic LML notation. Bob has expertise in RC Car Research, a salary of 48k and does not have a PhD.

The user-defined form-based visualizer which is applied to Bob is shown in the center top of Figure 7.3. The FormVisualizer contains a Form which receives its title from a CalculatedAttribute, which sets the title attribute of Form to the expression retrieving the name of the currently displayed employee. The breadcrumb of the Form is configured to be visible (showBreadcrumb=true) and to display the name of the currently displayed employee (attribute=name) through the BreadcrumbConfiguration owned by the Form. The grid layout of the Form has

one column (columns=1). In this grid layout there are two Groups, one displaying General Information (title=General Information) and arranging its content in two columns (columns=2) and one displaying Operations in its title (title=Operations).

The first widget placed in the General Information group is an Attribute mapped to the employee name (attribute=name). The second widget maps to the salary attribute (attribute=salary) and the third to the boolean hasPhD attribute. A BooleanAttribute is used to configure PhD as a label instead of hasPhD (label=PhD) and to use a checkbox to select whether the employee has a PhD (checkbox=true). The last Attribute maps to the expertise attribute.

In the Operations group there is only one Action which is of kind remove and has its label set to Delete Employee. The grabHorizontalExcess attribute is set to true and the horizontalAlignment attribute is set to end to right-align the button in the group.

The effect of this user-defined form-based visualizer is shown at the bottom right of Figure 7.3. The form displays Bob, the employee name, as its title. Also the breadcrumb is visible and shows the employee's name attribute value, Bob, as the currently selected model element. The General Information group is located below the breadcrumb containing text boxes for Name, Salary and Expertise. One checkbox is included to represent whether or not the employee has a PhD.

The operations group at the bottom of the form displays the Delete Employee button as the only operation available on the current employee. Pressing this button deletes the currently selected employee.

7.3 Form-based Weaving Model

The form-based format's weaving model, displayed in Figure 7.4, consists of five metaclasses — WeavingModel and WeavingLink with its three subclasses TraitWeavingLink, ExpressionWeavingLink and CompoundWeavingLink. The WeavingModel stores the WeavingLinks which establish a weaving between the model element displayed (target) and the widget displaying the information in the form-based language. The trait attribute of TraitWeavingLink determines which trait of the target is displayed and the ExpressionWeavingLink holds the expression used to calculate the text displayed in the widget together with the model

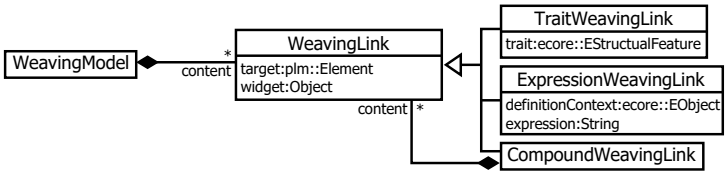


Figure 7.4: Form-based weaving model.

element on which the expression is defined (definitionContext). CompoundWeavingLinks are used when more than one model-element is displayed in a widget which is for example the case when displaying all connected clabjects in a list.

The available operations for manipulating the form-based weaving model are not described further because the weaving models created for form-based languages are trivial since they only depict one model element at a time. The weaving models are therefore built per model element and do not store any layout which could be influenced by a language user. Even when the delete or add operations are executed a completely new weaving model for the newly displayed model element is created which is independent of all previous weaving models. Additionally, both ends of the weaving model are directly addressable via pointers in memory making the implementation of editing functions trivial.

7.3.1 Form-based Weaving Model Example

An example of a user-defined, form-based language's weaving model is shown in Figure 7.5. The lower left of the figure shows an excerpt of the data underlying the form-based language visualized in a tree structure. The company example from previous chapters showing the Quality Toys Inc. is chosen. The bottom right of the figure shows Bob, an employee working in the Toy Research department of Quality Toys Inc. in a form-based view. The breadcrumb at the top shows the location of Bob in the underlying model. It has been configured to display the name attribute of companies, departments and employees in the breadcrumb. Below the breadcrumb, General Information about the selected employee (i.e. Bob) is shown. Bob has a Name set to Bob and a Salary set to 48k, does not have a PhD and has expertise in RC Car Research. Below the General Information group the Delete Employee button is placed in the Operations

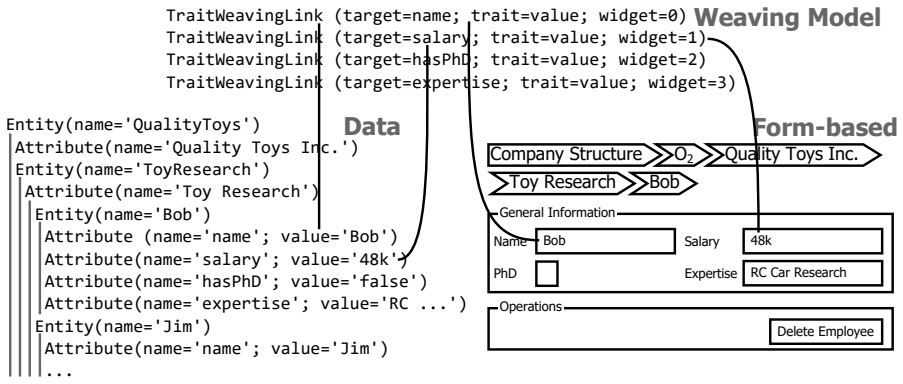


Figure 7.5: Form-based weaving model for the company structure modeling language example.

group. By pressing this button the currently selected employee (here Bob) can be deleted.

The Weaving Model instance at the top center of Figure 7.5 connects the Data and its Form-based visualization. For each widget representing an ontological attribute one TraitWeavingLink exists. All TraitWeavingLinks point to the value trait of one ontological attribute of Bob. In the figure, the attribute to which the TraitWeavingLinks point are identified by their name and the widget to which a TraitWeavingLink points is defined by the order of occurrence from left to right and top to bottom. In a tool, however, TraitWeavingLinks would address their targets and widgets through pointers to memory locations. For visualization purposes the TraitWeavingLinks of the name and salary attributes are indicated through solid lines.

Chapter 8

Deep Constraint Language for Deep Visualization

All of the visualization definition languages described in the previous chapters rely on constraints expressed in (deep) constraint languages to support visualization. More specifically, deep constraint languages are used to define expressions for: 1. calculating visualizer attributes, 2. defining the application conditions of aspects and 3. calculating values displayed to the user etc. In this chapter the challenges arising in the definition of deep constraint languages to support deep visualization are described and solutions to the different problems are discussed. This chapter also presents the deep constraint language developed in this thesis to support multi-format visualization.

8.1 Application Modes of Constraints in Deep Models

In two-level modeling technologies like the OMG's MOF, constraints are always defined on the fixed metamodel deployed into a modeling tool and are then executed on the metamodel instances. In their book on OCL, Warmer and Kleppe [238] refer to the metamodel types deployed in the modeling tool as *contextual types* and to the instances at the instance level as *contextual*

instances. In OCL, a constraint is defined on a contextual type and executed only on the contextual instances. However, this approach requires a clear separation of classes and instances in a modeling architecture. The deep modeling approach, however, blurs the distinction between classes and objects by replacing them with the notion of clabjects. A clabject can play the role of a type (i.e. a class) and an instance (i.e. an object) at the same time.

This blurring of the distinction between type level modeling and instance level modeling is clarified by the deep visualization mechanisms for deep models described in the previous chapters which simultaneously allow the definition of a user-defined visualization on a clabject and then visualizing this clabject in this user-defined visualization. This requires that constraints are not only executable on instances but also on the type through which these instances are defined. This lack of any distinction between types and instances makes the definition of contextual type and contextual instance problematic.

Another problem with this OCL terminology is that in a deep visualization setting it is not only necessary to apply constraints to direct instances at the level below the constraint definition but also on all following instance levels that contain instances more than one ontological classification level away from the type on which the constraint is defined, i.e instances of instances etc.

For these reasons, in [124] the alternative terms *definition context* and *execution context* were introduced to avoid this problem. These definitions do not rely on the distinction between types and instances but depend on the role a model element plays in the life cycle of a constraint. The definition context is the clabject on which a constraint is defined and the execution context is the clabject on which a constraint is executed. This definition allows a clabject to serve as the definition and execution context at the same time in contrast to the OCL which clearly requires contextual types and contextual instances to be separated and exist at two different classification levels. Moreover, this definition allows a constraint to be applied to instances more than one level away from the constraint definition because there is no need for the execution context to reside one level lower than the definition context.

Having introduced the notions of definition context and execution context the next question is how the extension of the execution context (i.e. set of all model elements on which a constraint is executed) is determined. A prag-

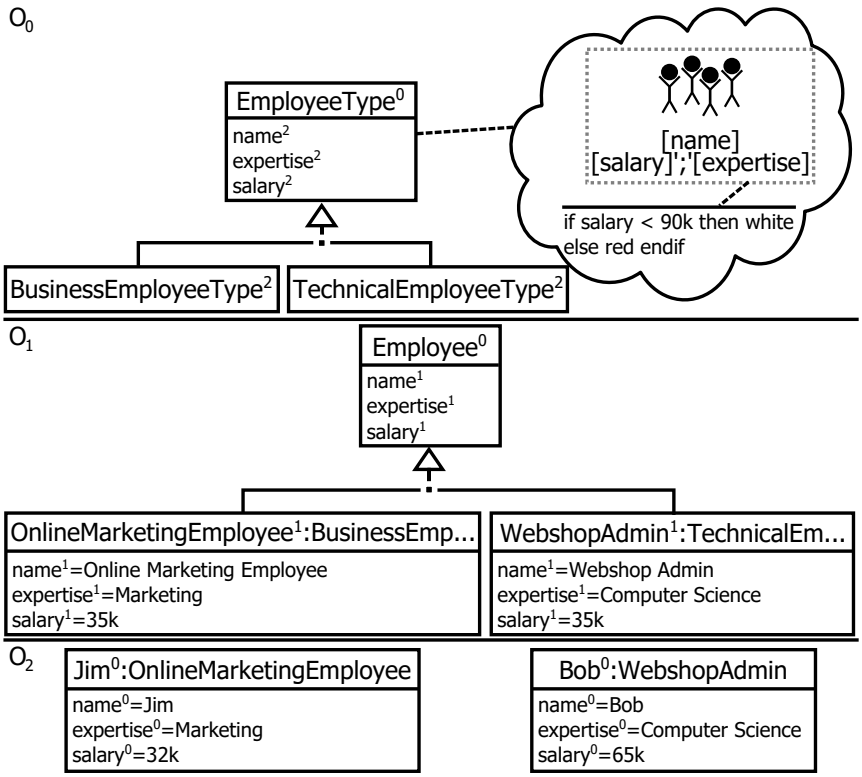


Figure 8.1: Constraint-defined background color of EmployeeType.

matic definition of *execution context extension* was adopted in this thesis which supports the need to define deep constraints for deep visualization. A deep visualization is applicable to the model element at which it is defined, that model element's subclasses and the whole classification tree derived from the model including subclasses and instances of subclasses at intermediate levels. Therefore, it makes sense to define all these clabjects on which a deep visualization is executed as the execution context extension of a constraint supporting deep visualization. Nevertheless, when defining the static semantics of a modeling language it is also necessary to have different, configurable execution context extensions. A discussion of this topic is provided in [19, 124].

Figure 8.1 shows an excerpt of the diagrammatic company structure modeling language. The visualizer of EmployeeType defines the group of stickmen

symbol used to represent EmployeeTypes in the previous chapters. The gray, dotted rectangle represents the model element's background which is defined through the constraint located at bottom of the visualizer. The constraint returns white as the background color for all EmployeeTypes with a salary lower than 90k and red as the background color for all other EmployeeTypes.

The definition context of the constraint is EmployeeType and the model elements which appear in the execution context extension are: 1. EmployeeType itself, 2. all subclasses of EmployeeType (BusinessEmployeeType and TechnicalEmployeeType), 3. the direct and indirect instances at the following level (OnlineMarketingEmployee and WebshopAdmin) and 4. the direct and indirect instances of the direct and indirect instances of EmployeeType (Jim and Bob). This large execution context extension allows the user-defined visualization attached to EmployeeType to be applied across all ontological classification levels in a deep model. A modeler can use the user-defined language, for example, to add new subtypes of EmployeeTypes at level O_0, create company-specific employee profiles at level O_1 or populate the O_2 level with actually existing employees in a company. At all levels, these clabjects will be rendered by the symbol attached to EmployeeType and their background will be red if they have a salary value higher than 90k.

8.2 Support of OCA-based Deep Modeling

Another difference between deep constraint languages and traditional constraint languages of the kind supported in the OMG's modeling infrastructure is that in deep modeling a clabject is classified by up to two types. One type, analogous to an object's type in the OMG's infrastructure, is the so-called ontological type residing one ontological level above the classified clabject. The other is the linguistic type which classifies all model content from a deep modeling language or tool point of view. Both dimensions contribute attributes (ontological attributes and linguistic traits) to clabjects which are useful for defining constraints. However, attribute names from one dimension can clash with attribute names from the other dimension. For instance, a modeler could define an ontological attribute *level* to store the level of a customer within a loyalty program (e.g. silver or gold member) but this would clash with (i.e.

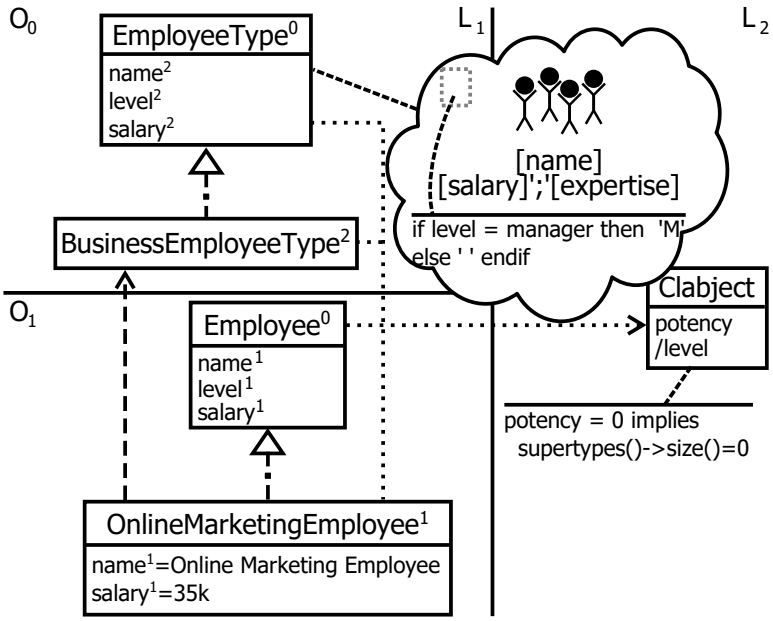


Figure 8.2: Constraints on the linguistic and ontological dimensions.

have the same name as) the predefined linguistic attribute (i.e. trait) *level* which points to the level containing the clabject (customer). A deep constraint language needs to allow such naming clashes to be disambiguated by clarifying which dimension it is referred to in an expression.

When a clabject has types from both dimensions it should also be possible to define constraints on both of them. Constraints on linguistic types effect their instances across all ontological levels. For example, it is possible to prohibit a clabject from having subtypes when it is located at a certain level or to define an upper bound on the potency values of clabjects in a deep model. Constraints on ontological types constrain only the ontological instances as previously described. However, it can be advantageous to access linguistic traits and operations when defining ontological constraints or the other way around. Hence, an OCA-aware, deep constraint language needs to support access to attributes/traits and operations from both dimensions in the definition of constraints.

Figure 8.2 shows an example of constraints affecting the linguistic and

ontological dimension. A linguistic constraint is defined on Clabject at the L_2 level that prohibits clabjects with potency 0 from participating in inheritance relationships. The ontological constraint on EmployeeType in O_0 states that the visualization of EmployeeTypes is dependent on the level attribute. If the level is set to manager an M is displayed in the upper right corner of the visualization, otherwise nothing is displayed at this location. This constraint highlights the ambiguity problem since it is not obvious for a constraint execution engine whether the ontological level attribute of EmployeeType or the linguistic trait of Clabject is meant in the example.

In this thesis the definition context is always assumed to be in the ontological dimension, because this is always the case for constraints used to support deep user-defined visualization. Constraint definitions in the linguistic dimension are, thus, out of scope of this work even though they may be important in other usage scenarios.

8.3 Definition of Constraints at Intermediate Levels

In the context of deep visualization a user-defined language can span more than one level. This is the case when instances provide aspects for user-defined type model visualizations. A possible problem when defining visualizations spanning more than one classification level is that on intermediate levels, connections can be further split and refined through the concept of connection diversification [20]. When combining the concept of connection diversification with user-defined language definitions spanning more than one level, it can happen that constraints at intermediate levels need to refer to navigations defined at higher classification levels that are not available at the level of the definition context.

An example of such a user-defined language definition spanning two levels in a setting where connection diversification is employed is shown in Figure 8.3 on the example of the diagrammatic company structure modeling language. At the highest level, O_0, four ontological types are defined — the abstract type EmployeeType and its subclasses BusinessEmployeeType, StudentEmployeeType

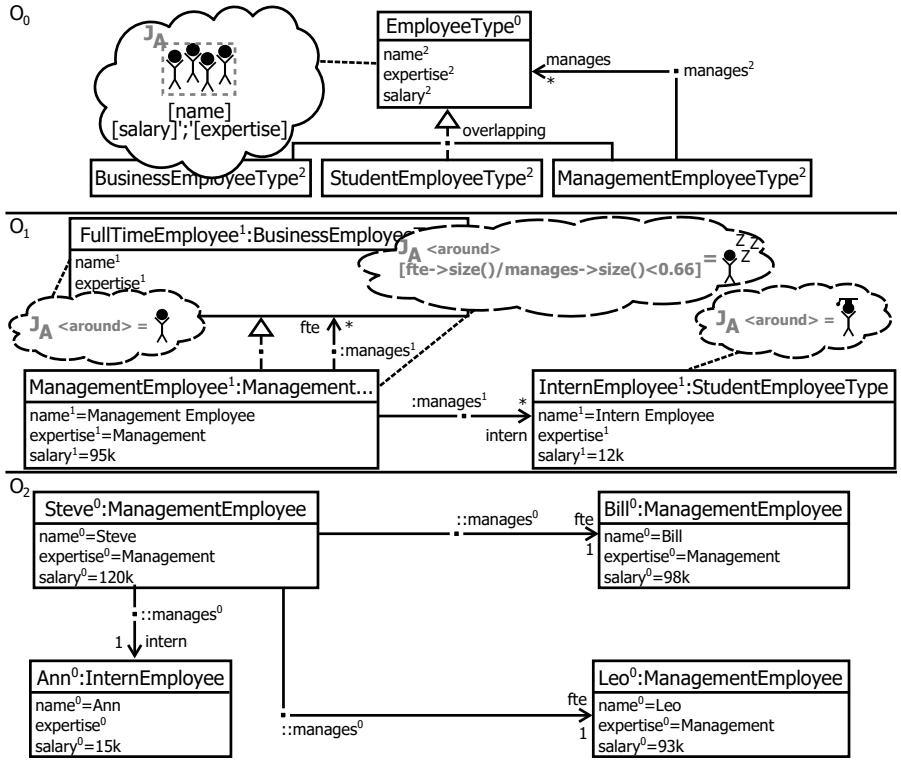


Figure 8.3: Aspect with condition at O_1 using a navigation defined on O_0.

and ManagementEmployeeType. ManagementEmployeeTypes manage an unlimited number of EmployeeTypes. As in previous examples EmployeeTypes are visualized through a group of stickmen and the name, salary and expertise attributes printed at the bottom.

At the intermediate level, O_1, three EmployeeTypes are defined — InternEmployee and FullTimeEmployee with its subclass ManagementEmployee. These three classes provide aspects to modify the icon displayed for visualizing the instance level (dashed border of visualizers), O_2. InternEmployees are visualized as a stickman wearing a square academic hat and FullTimeEmployees are visualized as a plain stickman. ManagementEmployees are also rendered as a plain stickman which is inherited from their supertype, FullTimeEmployee. Their aspect, however, implements the domain rule that the number of InternEmployees managed by a manager should not be more than one third of all managed

employees. If this rule is not followed, the aspect showing a stickman with Zs next to its head is applied.

To realize ManagementEmployee's visualization, its Aspect is enriched with a condition which must hold true for the aspect to be applied. To be able to distinguish between the managed FullTimeEmployees and InternEmployees the manages connection from level O_0 is diversified (i.e. split) into two connections with the connection end names fte pointing to FullTimeEmployee and intern pointing to InternEmployee. The constraint for determining the percentage of managed FullTimeEmployees divides the number of managed FullTimeEmployees by the number of all managed employees. For this purpose the size of the set of all FullTimeEmployees reachable via the fte navigation from O_1 is divided by the size of all managed employees reachable via the manages navigation from O_0.

In the example of Figure 8.3 the constraint is not ambiguous because the manages navigation exists on the O_0 level only and the fte navigation exists on the O_1 level only. This does not always have to be the case in more complex modeling scenarios. In the example it is also feasible for a ManagementEmployee to be connected to all managed FullTimeEmployees via a manages navigation and only to InternEmployees via an intern navigation. In such a scenario it would not be obvious whether the manages navigation used in the aspect defined at O_1 refers to the navigation at O_0 or O_1. Hence, a syntax offering disambiguated navigation access to navigations throughout all ontological classification levels of a deep model is needed.

The second source of issues when defining constraints at intermediate levels in a deep model is the lack of deep classification operations for checking the ontological type of a clabject, casting a clabject to another ontological type or retrieve the ontological instances of a clabject. In OCL all these operations work only across two levels. An instance can be checked for its type at the metamodel (e.g. *oclIsTypeOf()*), can be cast into another type from the metamodel (e.g. *oclAsType()*) and instances from the instance level of a type can be retrieved (e.g. *allInstances()*). However, as mentioned previously, these OCL operations work only on a pair of classification levels. To support deep visualization, therefore, these operations have to be extended to support deep modeling or new operations have to be added.

8.4 A Deep Constraint Language Supporting Deep Visualization

To keep the learning curve for modeling language engineers low, the deep constraint language introduced to support the deep visualization approach presented in this thesis is based on the OCL. The deep OCL variant [19, 124] has also been successfully used to extend the ATL [122] transformation language for deep modeling [25]. However, this section focuses on the OCL enhancements needed to support deep visualization scenarios. These are extensions to OCL in the three previously outlined problem areas: application modes, the support of OCA-based deep modeling and the definition of constraints on intermediate levels.

8.4.1 Constraint Application Modes for Deep Visualization

The default application mode for deep OCL constraints used in deep visualization is an execution extension that spans the whole deep execution extension described previously, i.e. a constraint is executed at the level on which it is defined and at all following levels. This application mode ensures that visualization definitions uniformly work across all visualization levels. However, this application mode creates difficulties when navigating over connections in a constraint language. When executing a navigation over a connection there is no guarantee that all connection ends have a cardinality with an upper bound of *one* leading to a single clabject as navigation result because, by definition, a cardinality constraint is satisfied when the sum of all lower and upper cardinalities of a connection's instances are within the range defined by the type. Hence, it can happen that when executing a constraint a navigation over a connection which has a cardinality range (e.g. 1..4) or an unlimited cardinality (i.e. *) is performed. The problem also arises when executing the constraint on the definition context, since the level typing other deep model elements is very likely to have connection cardinalities with upper bounds different to *one*. Clear semantics for navigation in such scenarios needs to be established to keep the definition and execution of constraints deterministic.

The issue is further illustrated in Figure 8.4 in which the constraint attached to ManagementEmployeeType navigates over the manages connection to EmployeeType. As part of a visualizer defined on ManagementEmployeeType this constraint is executed on all three ontological levels — O_0, O_1 and O_2. To determine the result on the intermediate levels where the connection ends are renamed, the type relationships between connection ends indicated as dashed arrows are used to retrieve the originally defined connection end name. These typing relationships are transitively followed until the executed constraint's definition context is reached. The only level at which the result is obvious is O_2 with the query result {Ann,Bree}, both are connected with cardinality 1 to Steve located in the classification tree of ManagementEmployeeType at which the constraint is defined. This query is straightforward to resolve because all connection cardinalities are 1. At the other levels, however, there are no clear statements about how many model elements are connected. The 0 to * multiplicity constraints defined for the manages navigation of ManagementEmployeeType and its instance at ManagementEmployee leaves the number of connected clabjects open.

There are six options for resolving this problem which depend on how cardinalities with upper bounds different to *one* are handled: 1. return a single clabject, 2. return a set with a single clabject, 3. return a number of clabjects equal to either the upper or lower bound, 4. return a statistically estimated number of clabjects, 5. do not execute the constraint and 6. distinguish between instance and type level navigation.

Single Clabject Returning a single clabject when a cardinality's upper bound is higher than *one* conforms to the definition of cardinalities making a statement about the instances of a connection. In cases where more than one clabject is reachable via the same navigation (i.e. more than one connection using the same navigation name is defined), a set containing each clabject exactly once is returned. In other words, a navigation always returns what is actually modeled in terms of connections. The problem with this solution, however, is that set operations such as size() do not run on a single model element because they are only defined for collections. Hence, this option is not compatible with the requirement that constraints should be applicable across

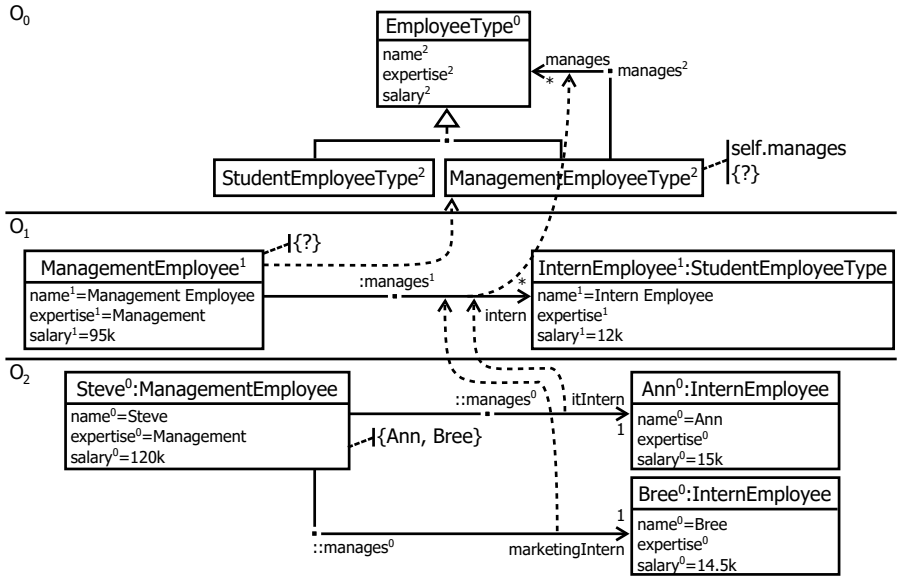


Figure 8.4: Deep navigation over the manages connections of ManagementEmployeeType.

all classification levels.

Constraint Example 8.4.1 (Single Clabject)**.**

1 *definition context* ***ManagementEmployeeType****: self.manages*
2 *execution context* ***ManagementEmployeeType*** ⇒ *EmployeeType*
3 *execution context* ***ManagementEmployee*** ⇒ *InternEmployee*
4 *execution context* ***Steve*** ⇒ *{Ann, Bree}*

Constraint Example 8.4.1 defines a navigation over the manages connection in the context of ManagementEmployeeType as shown in Figure 8.4. The first line of the example defines ManagementEmployeeType as the definition context (definition context ...) and the navigation over the manages connection (self.manages). The following lines first define on which clabject the constraint is executed, e.g. execution context ManagementEmployeeType, followed by the execution's result, e.g. ⇒ EmployeeType in line two. For the first two ontological levels, ManagementEmployeeType and ManagementEmployee, the navigation over manages results in a single model element (EmployeeType and InternEmployee) because cardinality upper bounds different to *one* are present. On the lowest

level, O_2, only cardinalities with upper bounds of *one* and two instances of the manages connection exist, so the result is a set of two clabjects ({Ann, Bree}).

Set with a Single Clabject This approach is basically equivalent to the *single clabject* approach but if the cardinality's upper bound is higher than *one* it wraps the result within a set. If more than one clabject is reachable via a navigation they are wrapped within a set which is identical to the single clabject approach. Returning a set with one clabject conforms with the intuition that cardinalities make a statement about instances only. Furthermore, set operations can be applied to such results and, hence, a constraint can be applied across an unlimited number of classification levels. The result of a query is also easy to predict. Here, Constraint Example 8.4.2 in the context of Figure 8.4 returns a set with one clabject when executed on the types of the first two ontological levels, ManagementEmployeeType and ManagementEmployee, on which the manages connection defines cardinalities with upper bounds higher than *one*. The results are {EmployeeType} for ManagementEmployeeType at level O_0 and {InternEmployee} for ManagementEmployee at level O_1. On the lowest ontological level again a set with two employees is returned {Ann, Bree}.

Constraint Example 8.4.2 (Set with Single Clabject)**.**

1 *definition context* ***ManagementEmployeeType****: self.manages*
2 *execution context* ***ManagementEmployeeType*** $\Rightarrow$ *{EmployeeType}*
3 *execution context* ***ManagementEmployee*** $\Rightarrow$ *{InternEmployee}*
4 *execution context* ***Steve*** $\Rightarrow$ *{Ann, Bree}*

In the example, this approach would cause problems on the lowest level, O_2, if Steve was only connected to Ann because it would return Ann as one single atomic model element, not wrapped up into a set. Hence, set operations valid on all intermediate levels would not be valid on the instance level anymore. To solve this issue the cardinality of the navigation on which the constraint is originally defined is taken into consideration. If this cardinality's upper bound is higher than *one*, which is the case in this example, a set with one element is returned regardless of whether the execution context is connected with one model element only having a cardinality with an upper bound of *one*. This principle is applied in all following approaches in order to maintain the validity of set operations across all classification levels.

Upper or Lower Bound When returning the upper or lower bound number of clabjects, the model element is duplicated as many times as the upper or lower bound defined for the navigation. This design decision does not conform to the intuition of only applying cardinalities to the number of connection instances but also enables application of constraints to the type level on which they are defined. It is necessary to decide which option to apply, either upper or lower bound, and to apply it uniformly to all constraints in the model so that the results of constraints are predictable by the modeler. Using the upper bound is problematic for the unlimited *star cardinality* because executing set operations on a set with an undefined number of elements is not supported by the OCL at the time of writing. Hence, the lower border is preferable because this problem does not exist since the lower border is always well defined. With this choice constraints are applicable across all classification levels and the results of constraints are predictable. In Constraint Example 8.4.3 it was decided to use the lower bound to calculate the navigation result, because in O_0 and O_1 two *star* potencies with an unlimited upper bound are present. Hence, the result of the navigation over manages on ManagementEmployeeType and ManagementEmployee is an empty set conforming to the *star* potency's lower bound of zero. Again, for Steve a set with two clabjects ({Ann, Bree}) is returned as a consequence of the lower bound of *one* of the cardinalities present at (O_2).

Constraint Example 8.4.3 (Upper Or Lower Bound)**.**

1 *definition context* ***ManagementEmployeeType****: self.manages*
2 *execution context* ***ManagementEmployeeType*** $\Rightarrow$ *{}*
3 *execution context* ***ManagementEmployee*** $\Rightarrow$ *{}*
4 *execution context* ***Steve*** $\Rightarrow$ *{Ann, Bree}*

Statistical Result Estimation The result of a constraint using a navigation can be estimated using statistical methods. An example of a probabilistic variant of OCL is P^2AMF [116], which for example can define derived attribute values based on a statistical probability function. Two approaches would be possible for such a scenario, either to have the modeler pick a statistical distribution underlying the navigation data or to have the constraint execution engine analyze the whole content of the deep model and return a corresponding

result. These solutions would work across all classification levels, but in the first case would require advanced statistical knowledge by a modeler to pick the right distribution and parameters. In the second case, navigations would return results which heavily depend on the content of a deep model and, thus, would be hard to predict when using the model for further modeling. Constraint Example 8.4.4 uses a simple prediction method to calculate the result of the navigation by calculating the average number of instances existing at all following levels. The calculation results in three clabjects for the level O_0 $((2 + 3)/2 = 2.5)$ and also for level O_1 $(3/1 = 3)$. Thus, a bag duplicating EmployeeType three times is returned at level O_0 for ManagementEmployeeType and at level O_1 for ManagementEmployee a bag containing InternEmployee three times is returned. At the lowest level a bag containing Ann and Bree is returned for Steve because two connections with a cardinality having an upper bound of *one* are connected to Steve.

Constraint Example 8.4.4 (Statistical Result Estimation)**.**
1 *definition context* ***ManagementEmployeeType****: self.manages*
2 *execution context* ***ManagementEmployeeType*** $\Rightarrow$ *{EmployeeType, EmployeeType, EmployeeType}*
3 *execution context* ***ManagementEmployee*** $\Rightarrow$ *{InternEmployee, InternEmployee, InternEmployee}*
4 *execution context* ***Steve*** $\Rightarrow$ *{Ann, Bree}*

No Execution A very conservative approach is to detect any ambiguous navigations, such as navigations with a cardinality upper bound higher than *one*, and instead of attempting to evaluate the result of the navigation a default value is returned by the whole constraint or parts of the constraint. The chosen default value can be *false*, which is equal to ignoring conditional visualizations with a constraint attached, or *true*, which is equal to ignoring conditions in deep visualizations. When executing constraints which do not return a boolean value but calculate a value (e.g. text for a label) the default could be to return an empty set as the result of the navigation. This approach of not executing an ambiguous navigation allows the constraint to be applied across all classification levels and is very easy to predict. Not executing constraints containing navigations with an upper bound cardinality different to *one* results

in no returned value for the constraint executed on ManagementEmployeeType and ManagementEmployee in Constraint Example 8.4.5. If these statements would be part of a conditional constraint for deep visualization, the execution would stop at this point and return *false*. The navigation from Steve returns a bag containing Ann and Bree.

Constraint Example 8.4.5 (No Execution)**.**

1 *definition context* ***ManagementEmployeeType****: self.manages*
2 *execution context* ***ManagementEmployeeType*** $\Rightarrow \emptyset$
3 *execution context* ***ManagementEmployee*** $\Rightarrow \emptyset$
4 *execution context* ***Steve*** $\Rightarrow$ *{Ann, Bree}*

Instance and Type Level Navigation The *instance and type level navigation* approach is a hybrid approach composed of the previously described *set with single clabject* and *upper or lower bound* approaches. If the definition context and execution context are equal, the navigation is viewed as a type level navigation and the *set with single clabject* approach is applied. Thus, a navigation returns exactly what is modeled, resulting in a set with all clabjects that are actually connected by the navigated connection in case of a cardinality with an upper bound different to *one* and a single clabject in case of a cardinality upper bound of *one* (if more than one clabject is reachable via such a navigation a set with all connected clabjects is returned). When the definition context is on a level above the execution context, an instance level navigation is performed by applying the *upper or lower bound* approach. Hence, a bag repeating the connected model element either as often as the lower or upper bound is returned as the result of the navigation. By applying the *upper or lower bound* approach on the execution context where an instance level navigation takes place the cardinalities at the definition context level are treated as statements about the instances of the definition context on which the constraint is executed because the cardinalities at the instance levels are defined to satisfy their type's cardinalities. This option for resolving the problem of connection cardinalities is the one used in the rest of this work because this is seen as the best way of treating cardinalities as statements on instances and on the other hand being able to execute a constraint across all ontological levels.

When applying this approach in the context of Constraint Example 8.4.6 the constraint on ManagementEmployeeType is executed from a type level navigation point of view in line 2, because the definition and execution contexts are identical. This returns a set with the actually connected EmployeeType without paying any attention to the modeled cardinality's lower or upper bound. The other navigations (line 3 and line 4) are viewed as instance level navigations and therefore return the lower bound of the cardinalities which is an empty set for ManagementEmployee and a set containing Ann and Bree ({Ann, Bree}) for Steve.

Constraint Example 8.4.6 (Instance and Type Level Navigation)**.**
1 *definition context* ***ManagementEmployeeType****: self.manages*
2 *execution context* ***ManagementEmployeeType*** $\Rightarrow$ *{EmployeeType}*
3 *execution context* ***ManagementEmployee*** $\Rightarrow$ *{}*
4 *execution context* ***Steve*** $\Rightarrow$ *{Ann, Bree}*

8.4.2 (Re)Classification Operations

OCA-based deep modeling also has an impact on (re)classification operations (i.e. type checking, type casting and instance retrieval) in a deep constraint language. (Re)classification operations applied to a clabject are not only of use across one type/instance classification level pair but across all ontological classification levels in a deep model. For this purpose, the existing OCL instance query operations (*allInstances()*), type checking operations (*isKindOf()*, *isTypeOf()*) and type casting operations (*asType()*) have to be extended to work across more than one ontological type/instance level pair. Below, we first introduce suitable terminology related to typing in a deep model, clearly define the main concepts in first order logic and finally present deep (re)classification operations based on these definitions.

The instances of a clabject in deep modeling can be distinguished between *instances* and *deep instances*. Instances exist at the level directly below the clabject whereas deep instances exist across all classification levels determined by the transitive closure of all classification relationships, i.e all instances, including instances of instances etc. Additionally, a distinction can be made between direct and indirect (deep)instances. Direct (deep)instances exclude

instances of subtypes whereas indirect (deep)instances include instances of subtypes only. The following sections discuss a formalization of these concepts in first order logic extended by the transitive closure operator TC().

Definition 8.4.1 shows a formal definition of the direct instance relationship (IsDirectInstance(t,i)) between an instance i and a type t. Direct instances of t are instances which are modeled as instance of t only and not as instance of one of t's subtypes. Four functions are used to support the definition of this relationship. The function classifications(x) maps each clabject x to the classifications it participates in (*clabject* → *classification*). The function typeEnd(x) maps a classification to the clabject located at its type end (*classification* → *clabject*). The instanceEnd(x) function maps classifications to the clabject at the instance end (*classification* → *clabject*). These previous three functions are used to construct the directInstancesOf(t) function which returns all instances of a clabject t (*clabject* → *clabject*). The instances of t are located at the instance ends of the classifications in which t participates minus t itself. In the function, t has to be removed from this set because in a deep modeling framework t can play the role of an instance that is located at the instance end of a classification relationship and as a type that is located at the type end of a classification relationship at the same time. The IsDirectInstance(t,i) relationship between an instance i and a type t exists if i is in the set of direct instances of t.

Definition 8.4.1 (IsDirectInstance(t,i))**.**

1 *classifications(x):= All classifications where x participates*
2 *typeEnd(x):= Clabject at the type end of classification x*
3 *instanceEnd(x):= Clabject at the instance end of classification x*
4 *directInstancesOf(t) = {instanceEnd(c) | c ∈ classifications(t)} \ {t}*
5 *IsDirectInstance(t,i):= i ∈ directInstancesOf(t)*

The definition of the indirect instance relationship (IsIndirectInstance(t,i)) is shown in Definition 8.4.2. Indirect instances are instances of the subtypes of a type excluding direct instances of that type. The previously defined function directInstancesOf(t) and all of its supporting functions are reused in this definition. Since indirect instances include subtypes, functions for querying those have to be added to the formalization. These functions are inheritances(x) (*clabject* → *inheritance*), superTypeEnd(x) (*inheritance* → *clabject*), subType-

End(x) (*inheritance* → *clabject*) and subTypes(x) (*clabject* → *clabject*) which are equivalent to the corresponding functions for classification retrieval with the exception that they work on inheritance relationships. An important difference between the classifications(x) and the inheritances(x) functions is that the inheritances(x) function not only returns the inheritance relationships pointing to subtypes of x (direct) but also the inheritance relationships pointing to subtypes of subtypes etc. (indirect). To retrieve the indirect instances of a type t the indirectInstancesOf(t) (*clabject* → *clabject*) function is used to return the instances of the subtypes of t. The indirectInstancesOf(t) function is then used to define the IsIndirectInstance(t,i) relationship between an instance i and its type t. An instance i satisfies the indirect instance requirement by being an instance of one of t's subtypes (i ∈ indirectInstancesOf(t)). If it is intended to make no distinction between direct and indirect instances the IsInstance(t,i) relation is applicable since it includes direct and indirect instances (IsDirectInstance(t,i) ∨ IsIndirectInstance(t,i)).

Definition 8.4.2 (IsIndirectInstance(t,i) and IsInstance(t,i))**.**

1 *inheritances(x):= Direct and indirect subtype inheritance relationships of x*
2 *superTypeEnd(x):= Clabjects at supertype end of inheritance relationship x*
3 *subTypeEnd(x):= Clabjects at subtype end of inheritance relationship x*
4 *subTypes(x):= {subTypeEnd(i) | i ∈ inheritances(x)}*
5 *indirectInstancesOf(t):= {directInstancesOf(s) | s ∈ subTypes(t)}*
6 *IsIndirectInstance(t,i):= i ∈ indirectInstancesOf(t)*
7 *IsInstance(t,i):= IsDirectInstance(t,i) ∨ IsIndirectInstance(t,i)*

The definitions so far have only included instances present at the classification level below the type's classification level. Deep direct instances and deep indirect instances include instances over the whole classification tree of a type. The definition of the IsDeepDirectInstance(t,i) function is shown in Definition 8.4.3. It uses the set of all classifications C which are directed edges (t,i) pointing from the type t to the instance i (i = instanceEnd(c)). The transitive closure on the set of all classifications (TC(C)) is then used to define the IsDeepDirectInstance(t,i) relationship. The TC() operation on a set adds all indirect paths to this set. If, for example, two classifications (a,b) and (b,c) are in the set of all classifications, the tuple (a,c) would be added to this set

by the TC operation, because there is a path from a to c via b. The IsDeepDirectInstance(t,i) relationship holds true if the tuple (t,i), where t is the type and i is the instance, is in the transitive closure of all classification relationships C. Being in the transitive closure means that there is a path of classification relationships from the type t to the instance i.

Definition 8.4.3 (IsDeepDirectInstance(t,i))**.**

1 *C:= {(t,i) | c ∈ classifications(t) ∧ i = instanceEnd(c)}*

2 *IsDeepDirectInstance(t,i):= (t,i) ∈ TC(C) (TC() - Transitive Closure)*

The definition of the *deep indirect instance* relationship (Definition 8.4.4) includes the set of all inheritance relationships I in addition to all classifications C. Inheritance relationships are directed edges pointing from their supertype x to their subtype y (y = subTypeEnd(i)). The IsDeepIndirectInstance(t,i) relationship includes the instances of subtypes of a type by using the union of the inheritance and classification relationships as input for the transitive closure function (TC(C∪I)). The transitive closure on the set of classifications C and inheritance relationships I includes all edges from a type t to an instance i reachable via inheritance and classification relationships. Since the definition of indirect instances excludes direct instances, all direct instances which are contained in the transitive closure over the classification relationships are subtracted (\TC(C)). Furthermore, all inheritance relationships (direct and indirect) are subtracted (\TC(I)) since there is no interest in any inheritance relationships. The IsDeepInstance(t,i) relationship is used in cases where the distinction between direct and indirect deep instances is irrelevant.

Definition 8.4.4 (IsDeepIndirectInstance(t,i) and IsDeepInstance(t,i))**.**

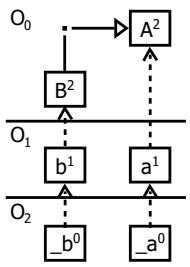
1 *I:= {(x,y) | i ∈ inheritance(x) ∧ y = subTypeEnd(i)}*

2 *IsDeepIndirectInstance(t,i):= (t,i) ∈ (TC(C ∪ I) \ TC(C) \ TC(I))*

3 *IsDeepInstance(x,y):=*

IsDeepDirectInstance(x,y) ∨ IsDeepIndirectInstance(x,y)

An example calculation for the IsDeepIndirectInstance(t,i) relationship is shown in Figure 8.5. In the example all relationships for which IsDeepIndirectInstance(t,i) holds true are calculated in the context of the diagram on the figure's left-hand side. It is important to note that in the formalization the pointing directions



(a)

```
1  C = {(A,a), (a,_a),(B,b),(b,_b)}
2  I = {(A,B)}
3  I∪C = {(A,B),(A,a),(a,_a),(B,b),(b,_b)}
4
5  TC(C) = C ∪ {(A,_a),(B, _b)}
6  TC(I) = I
7  TC(I∪C) = I ∪ TC(C) ∪ {(A,b),(A,_b)}
8  TC(I∪C)\TC(C) = TC(I∪C)\{(A,a),(a,_a),(B,b),(b,_b)}
9                = {(A,B),(A,b),(A,_b)} = X
10 X\TC(I) = {(A,b),(A,__b)}
11       = TC(C∪I)\TC(C)\TC(I)
```

(b)

Figure 8.5: Example of the IsDeepIndirectInstance(t,i) relationship.

of inheritance and classification relationships are reversed with respect to the standard LML rendering shown in the figure. The result of the calculation, shown in line 10 at the right side of Figure 8.5, is that b is a deep indirect instance of A and _b is a deep indirect instance of A.

The operations of the deep constraint language for instance retrieval, type checking and casting follow the semantics and naming schema of the previously introduced formal definitions. For each operation four methods divided into two dimensions are introduced, as shown in Figure 8.6. The operations are divided according to their deepness (classification) and directness (inheritance) properties. The deepness specifies whether an operation includes instances at the next classification level only or over all following classification levels in a deep model. The directness property specifies whether the operation also includes subtypes or is restricted to the type on which the operation is called. All operations that exclude subtypes are designated with the word direct in their name, and the others working on subtypes only have the word indirect in their name. If no distinction is made between direct and indirect, no special designation is given in the name. Operations which include all classification levels carry the word deep in their name, while all others have no special designation in their name. The application of this naming scheme result yields the following operations which are described in more detail in the following paragraphs: *classification checking* — isInstanceOf(), isDeepInstanceOf(), isDirectInstanceOf(), isDeepDirectInstanceOf(), isIndirectInstanceOf(), isDeepIndirectInstanceOf() —, *instance querying* — allInstances(), allDeepInstances(),

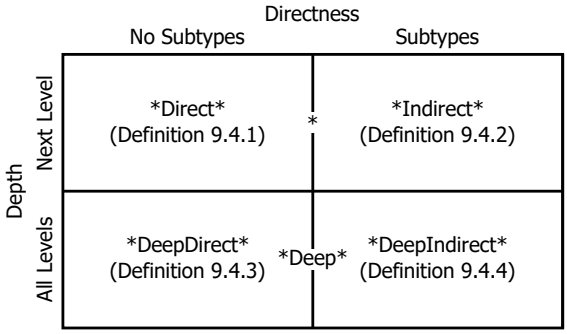


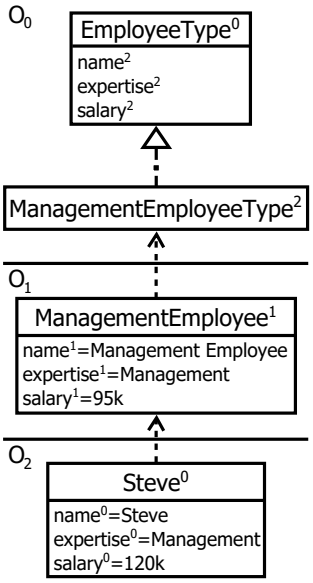
Figure 8.6: Naming scheme for (re)classification operations.

allDirectInstances(), allDeepDirectInstances(), allIndirectInstances(), allDeepIndirectInstances() — *type casting* — asType(), asDeepType().

Classification Checking An example of the classification checking operation is given on the company structure modeling language example in Figure 8.7. The table in (b) shows the different classification checking methods executed in the context of Steve. The model underlying the constraints is displayed in (a). It can be observed that the isInstanceOf() operation returning *true* for types at the immediate level above the instance (here ManagementEmployee) including their subtypes is equivalent to *oclIsKindOf()* in OCL. For all other clabjects in the classification hierarchy false is returned because these are more than one level away from Steve. If ManagementEmployee had subtypes *true* would also be returned for these by the isInstanceOf() operation.

Furthermore, the isDirectInstanceOf() operation returning *true* for types at the immediate level above the instance excluding subtypes of the type is equivalent to the *oclIsTypeOf()* operation in OCL. Hence, the operation returns *true* for ManagementEmployee and *false* for all other clabjects when executed on Steve. In this example the execution of the isInstanceOf() and isDirectInstanceOf() operations return the same result since Steve's type has no subtypes. If the operations were executed on Steve as an instance of a subtype of ManagementEmployee and ManagementEmployee as parameter, *false* would be returned by the isDirectInstanceOf() operation and *true* by the isInstanceOf() operation.

The deep versions of the isInstance() and isDirectInstanceOf() operations are is-



(a)

context Steve isInstanceOf(...)	
ManagementEmployee	true
ManagementEmployeeType	false
EmployeeType	false

context Steve isDeepInstanceOf(...)	
ManagementEmployee	true
ManagementEmployeeType	true
EmployeeType	true

context Steve isDirectInstanceOf(...)	
ManagementEmployee	true
ManagementEmployeeType	false
EmployeeType	false

context Steve isDeepDirectInstanceOf(...)	
ManagementEmployee	true
ManagementEmployeeType	true
EmployeeType	false

context Steve isIndirectInstanceOf(...)	
ManagementEmployee	false
ManagementEmployeeType	false
EmployeeType	false

context Steve isDeepIndirectInstanceOf(...)	
ManagementEmployee	false
ManagementEmployeeType	false
EmployeeType	true

(b)

Figure 8.7: Classification checking operations on the example of Steve.

DeepInstanceOf() and isDeepDirectInstanceOf(). These two operations are equal to their non-deep versions except that they can check for classification across more than one pair of classification levels. Hence, isDeepInstanceOf() returns *true* for ManagementEmployee, ManagementEmployeeType and EmployeeType, whereas isDeepDirectInstanceOf() returns *false* for EmployeeType as Steve is instance of a subtype of EmployeeType and not a direct instance of EmployeeType.

In addition to these four functions inspired by OCL, two classification-checking operations only for indirect instances are defined, isIndirectInstanceOf() and isDeepIndirectInstanceOf(). These return *true* only for indirect instances of a type. Thus, both the flat and deep versions return *false* for Manage-

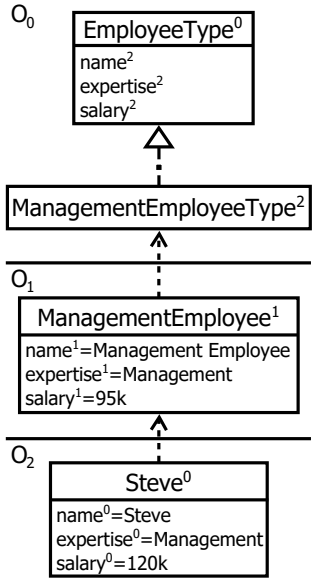
mentEmployee because Steve is a direct instance of ManagementEmployee. The flat version, isIndirectInstanceOf(), returns *false* for ManagementEmployeeType and EmployeeType since these are located more than one level above Steve. The deep version returns *false* for ManagementEmployeeType because Steve is a deep, direct instance of it and *true* for EmployeeType because Steve is a deep instance of one of EmployeeType's subtypes, and hence a deep, indirect instance of it.

Linguistic type checking can be invoked using the standard OCL type checking operations *oclIsTypeOf()* and *oclIsKindOf()* available from the metamodeling framework in which the linguistic metamodel is implemented. In order to check if Steve is an instance of the linguistic type clabject, it is necessary to first switch to the linguistic dimension and then invoke the *oclIsKindOf()* operation — *self._l_.oclIsKindOf(Clabject)*. Such an operation call on Steve returns *true* because Steve is a linguistic instance of the linguistic metamodel's clabject type.

Instance Retrieval Figure 8.8 illustrates the six instance retrieval operations available in a deep constraint language in the context of EmployeeType and ManagementEmployeeType. Both, the directInstances() and deepDirectInstances(), operations do not return a clabject when called on EmployeeType because EmployeeType only possesses indirect instances. In contrast, the indirectInstances() operation and its deep version, deepIndirectInstances(), return ManagementEmployee and additionally Steve in the case of the deep version. Querying for both indirect and direct instances through the instances() and deepInstances() operations returns ManagementEmployee for the flat version and additionally Steve for the deep version.

Executing the instances() and deepInstances() operations on ManagementEmployeeType returns the same values as previously described for EmployeeType. However, the operations for indirect and direct instance retrieval return the opposite results as for EmployeeType because ManagementEmployeeType is a subtype of EmployeeType. Hence, directInstances() returns ManagementEmployee and deepDirectInstances() additionally returns Steve. The indirectInstances() and deepIndirectInstances() operation return empty sets because ManagementEmployee has no indirect types at any level.

The deep instance retrieval operations presented so far return instances



(a)

context EmployeeType	
directInstances()	∅
indirectInstances()	{ManagementEmployee}
instances()	{ManagementEmployee}
deepDirectInstances()	∅
deepIndirectInstances()	{ManagementEmployee, Steve}
deepInstances()	{ManagementEmployee, Steve}

context ManagementEmployeeType	
directInstances()	{ManagementEmployee}
indirectInstances()	∅
instances()	{ManagementEmployee}
deepDirectInstances()	{ManagementEmployee, Steve}
deepIndirectInstances()	∅
deepInstances()	{ManagementEmployee, Steve}

(b)

Figure 8.8: Instance retrieval operation examples.

across all ontological levels or from the next ontological level. In some cases, however, it is desirable to only query instances at a specific level or range of levels. Therefore, the deep instance retrieval operations accept either one parameter specifying the distance to the level to query (e.g. all direct instances that are two levels away) or two parameters specifying the range of levels to query (e.g. all instances from the levels with distance two to four). The distance is always specified as a relative value with *one* specifying the next level and *star* (*) specifying the last level.

It is possible to retrieve linguistic instances (e.g. all clabjects in a deep model) by invoking the *allInstances()* OCL operation on the type of the linguistic metamodel. To get all instances of the linguistic type of Steve for example, one first switches to the linguistic dimension and then invokes the *allInstances()* operation (*self.__l__.allInstances()*). In the case of Steve, this would return all clabjects of the deep model including Steve, because the linguistic type of Steve is clabject.

Reclassification Two reclassification operations are offered by the deep constraint language — *asType()* and *asDeepType()*. The condition for a type cast to succeed is that the clabject to be cast is either an instance (*asType()*) or deep instance (*asDeepType()*) of the type into which it is to be cast. The *asType()* operation can cast a clabject into a type one ontological level above. Hence, in the example displayed in Figure 8.8, Steve can be cast into ManagementEmployee but not into ManagementEmployeeType. To cast Steve into ManagementEmployeeType the *asDeepType()* operation is used which can also cast Steve into ManagementEmployee and EmployeeType.

8.4.3 Constraints on Intermediate Levels using Higher Level Navigations and OCA

To support the definition of constraints on intermediate levels which refer to navigations defined at higher ontological classification levels a syntax modification is introduced to dynamically move the definition context of a clabject to a level on which the required navigations are available. To do this definition context move, the new definition context is enclosed in underscores (_) and is used in the same way as an attribute or operation access.

Constraint Example 8.4.7 (Definition Context Movement)**.**

```
1 definition context Steve: self._ManagementEmployee_.intern
2 execution context Steve ⇒ {Ann,Bree}
```

Constraint Example 8.4.7 shows an example in the context of Figure 8.9. The constraint is defined in the definition context of Steve. The intent of the constraint is to retrieve all interns who are managed by Steve. Steve, however, possesses two different ways of navigating to the managed interns (itIntern, marketingIntern). The currently defined navigations can be extended in future as needed, so for example an hrIntern navigation could be introduced if Steve starts managing interns in the human resource department. Hence, the intern navigation from level O_1 must be used to navigate to all interns. To use this navigation, a definition context move from Steve to ManagementEmployee is made (self._ManagementEmployee_) in the first line of Constraint Example 8.4.7, making the navigations of ManagementEmployee available — here intern only.

The intern navigation is then used to navigate to all interns managed by Steve. The result of this navigation, shown in line two, is a set containing Ann and Bree who are connected with Steve via instances of the connection defining the intern navigation.

A consequence of applying a definition context move is that a navigation can change from a *type level navigation* to an *instance level navigation* if the definition and execution context were equal before the move and are different after. This influences the result of a navigation over connections with a cardinality upper bound different to *one*. In the example shown here the navigation changes from a type level navigation to an instance level navigation through the definition context move. This is not a problem in this case because Steve is only connected via connections with cardinality upper bounds of *one*.

Orthogonal classification of model elements is supported by the option to select either the ontological or linguistic dimension for constraint definition in the deep constraint language. The default context for constraints is the ontological dimension because user-defined visualizations are defined within this dimension. To switch to the linguistic dimension the expression *__l__* is used in the same way as an attribute access. Once the switch to the linguistic dimension has been performed all following attribute calls etc. are performed from a linguistic point of view. To switch back to the ontological dimension the expression *__o__* is used.

Constraint Example 8.4.8 (OCA Support)**.**

1 *definition context* ***Steve****: self.__l__.getAllAttributes().value*
2 *execution context* ***Steve*** *⇒ {'Steve', 'Management', '120k'}*

3 *definition context* ***StudentEmployeeType****: self.__l__.getAllSupertypes().__o__.salary*
4 *execution context* ***StudentEmployeeType*** *⇒ {''}*

Constraint Example 8.4.8 demonstrates the seamless switching between the linguistic and ontological dimension in the context of the example shown in Figure 8.9. In the figure the linguistic metamodel (L_2) is placed on the right spanning all ontological classification levels (O_0 - O_2). The first line of the constraint example defines a constraint on Steve that first switches over to the linguistic dimension (__l__) to make all linguistic metamodel features of Clabject,

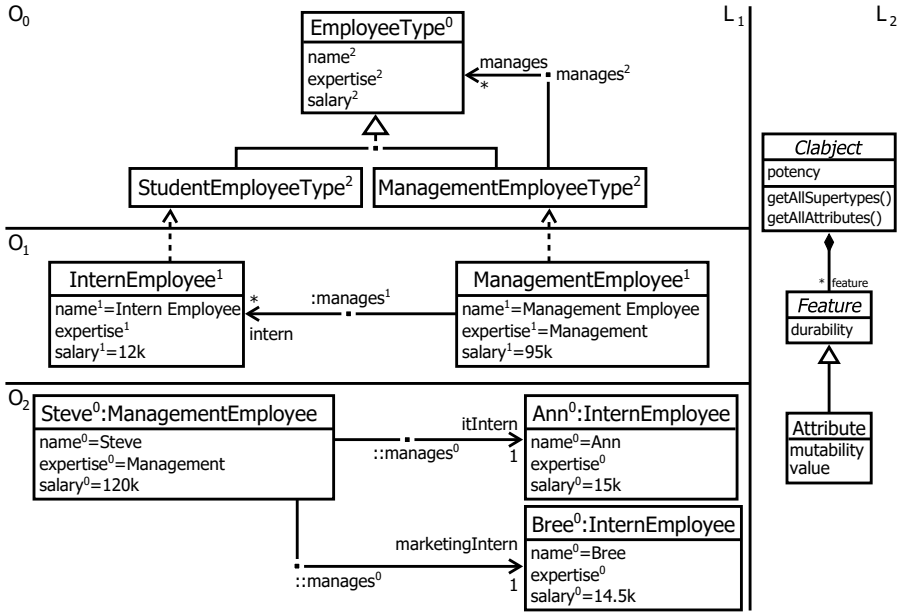


Figure 8.9: Constraint example in the context of the OCA.

Steve's linguistic type, available. Then the getAllAttributes() method of Clabject is used to retrieve the list of all attributes belonging to Steve. These are then queried for their values using the linguistic value trait. The second line of the constraint example displays {'Steve', 'management, '120k'} as the result of the executed query. The second constraint starting in the third line of the example defines a constraint on StudentEmployeeType. First, it queries all supertypes of StudentEmployeeType using the linguistic getAllSupertypes() operation and then switches back to the ontological dimension (_o_) to get the salary of all supertypes. The result of this query displayed in the last line of the constraint example shows an empty string because EmployeeType, the only supertype of StudentEmployeeType, does not have a salary value defined.

Chapter 9

Seamless Modeling

Deep modeling makes it possible for a modeler to work on several classification levels at the same time since all levels are treated equally and no deployment steps are required to make changes to one classification level available to the others. This feature, called seamless modeling below, gives significantly more flexibility to modelers than metamodeling approaches using a fixed metamodel level and a soft user model level. To change the metamodel and apply the changes to the deployed modeling tool, several steps have to be executed in such approaches. First, the modeling environment has to be switched into the metamodeling mode which often involves a switch from the tool in which the modeling language is used to the tool in which the modeling language is defined. Second, the new metamodel has to be supplied to the modeling environment which often involves manual effort such as deploying the new version of the metamodel including its accompanying tool to a central update repository and invoking the update mechanism of each deployed tool. Third, model evolution mechanisms have to be applied. When using deep modeling, the first two of these tasks are completely unnecessary since there is no difference between metamodeling and user modeling (i.e. a modeler can use all classification levels seamlessly). The model evolution problem, however, also exists in deep modeling. In fact, the problem is even more acute in deep modeling since it is much easier for modelers to make changes to more than one classification level. In contrast, in traditional modeling environments based on one clas-

sification level pair, the operational classification level can only be switched when the underlying modeling language is changed (e.g. from the user-defined language at M_1 to the metamodeling language at M_2). As a consequence, the vast majority of work is done at the user-model level.

Modeling across multiple classification levels also increases the impact of changes to a model because it is not longer just one classification level that is affected by a change, multiple classification levels can be affected in both the type and instance directions of the classification hierarchy at the same time. Figure 9.1 gives an example of the complexity introduced by a change in a model consisting of seven model elements. The language shows an excerpt of the company structure modeling language. Using this language, different CompanyTypes can be created. ITCompany, a type of company producing information technology (IT) related goods, and ToyCompany, a type of company producing toys, are created as two types of companies at level O_1. These are themselves instantiated with four concrete companies named Pineapple, Bananasoft, Supertoys and Boringtoys. After the whole model has been created, it is noticed that a company has to possess a taxID so the taxID attribute is added to the CompanyType clabject. This change, however, makes all instances and instances of instances of CompanyType invalid. To fix the model, all six clabjects in the classification hierarchy of CompanyType — ITCompany, ToyCompany, Pineapple, Bananasoft, Supertoys and Boringtoys — have to have their taxID attribute added manually. Moreover, the same manual effort would have to be performed when changes to potencies, durabilities, attribute data types etc. are performed. The effort of such manual changes grows as the number of inheritance relationships, classification levels and model elements increases, making the approach increasingly difficult to use in an iterative development process.

To handle the complexity of these knock-on changes when working at multiple classification levels, a good deep modeling tool should provide a so-called *emendation service* [16]. Such a service constantly watches for changes to the edited deep model at all ontological levels and automatically calculates the impact of these changes in terms of violations of the classification consistency rules, as e.g. defined by Kennel in [127]. If a violation of these rules is detected, the service attempts to correct the classification relationships con-

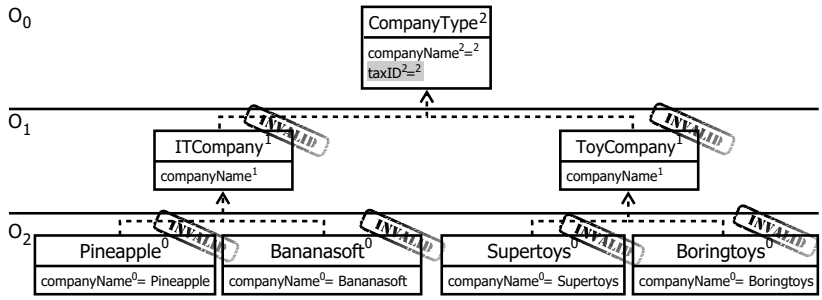


Figure 9.1: Impact of adding the taxID attribute.

cerned by executing changes to repair the affected parts of the model. This is done by suggesting update operations over the whole deep model and applying them if confirmed by the modeler.

The idea of applying evolution services to models is not new. Research on this topic has been performed in the area of metamodel (co)-evolution (e.g. [39, 107, 108, 199, 200]), ontology evolution (e.g. [132, 165, 173]) and deep modeling (e.g. [16, 56, 57]). The first research on model evolution in the context of deep models was performed by Demuth et al. in [57]. However, this only briefly covered the evolution of deep models as a side aspect of a deep modeling extension to Rational Software Architect [156] (at the time of writing known as IBM Rational Software Modeler). The most complete consideration of deep model emendation, upon which this work is based, is described by the author of this thesis in [16].

9.1 Used Formalism

The classification rules for the deep modeling approach used in this work as the foundations of the emendation service are defined by Kennel in [127]. The principles of the emendation service, however, also work with other formalizations of deep modeling such as that of Rossini et al. [204]. In the following, classification rules are expressed using first order logic and set theory. The *for-all* ($\forall$) and *exists* ($\exists$) quantifiers as well as the *element-of* ($\in$) operator are used from set theory. Furthermore, the logical *and* ($\wedge$), *or* ($\vee$), *implies* ($\implies$), *equal* ($=$) and *not-equal* ($\neq$) operators are used from propositional

Symbol	Description
C	The set of all clabjects (entities and connections)
c	A clabject from the set of all clabjects
potency(c)	The potency of c
c_t	The type of the clabject c, i.e. c_t = type(c)
connections(c)	The connections associated with c
attributes(c)	All attributes of c including inherited attributes
con	A connection from the set of all clabjects
con_t	The type of a connection, i.e. con_t = type(con)
mandatory(con_t,c)	Is connection con_t mandatory for c?
conforms(con_t,con)	Does connection con conform to its type con_t?
$\forall c \in C$	For each clabject c from the set of all clabjects C
$\exists c \in C$	It exists one clabject c in the set of all clabjects C

Table 9.1: Description of the used symbols.

logic, functions and predicates are used from first order logic. Table 9.1 gives an overview of the symbols and conventions used in the formalization. Capital letters represent the set of all instances of a linguistic meta type. For instance C refers to the set of all clabjects in a deep model. Lowercase letters refer to a single element in the set of all elements of a type. Hence, c refers to one particular clabject from the set of all clabjects C. Access to linguistic metamodel attributes (also known as traits) and methods is designated by a function named after the method name (e.g. $\mathsf{conforms}(\mathsf{con}_t,\mathsf{con})$) or trait name which is accessed (e.g. $\mathsf{durability}(\mathsf{c})$). An ontological type of a model element is designated by a t in the index (e.g. c_t) and ontological instances by an i in the index (e.g. c_i).

9.2 Deep Model Consistency

When a clabject is changed a deep model can cease to be well-formed. For a deep model to be well-formed, the content of all its levels has to be well-formed. This, in turn, depends on *classification correctness* and *inheritance correctness*, which can both be violated through a change to a deep model.

Definitions for these well-formedness properties are extensively introduced by Kennel in [127] and are shortly summarized in the following.

For a type to satisfy classification correctness, all instances of the type must be valid instances. To be a valid instance of a type, a clabject has to fulfill the following conditions: 1. the potency of the instance must conform to the potency of the type (Definition 9.2.1), 2. an instance has to have a conforming attribute for every attribute of the type (Definition 9.2.2) and 3. an instance must have at least one connection for every mandatory connection (i.e. lower multiplicity bound greater one) of the type the (Definition 9.2.3).

Definition 9.2.1 (Potency Correctness)**.** *Every clabject (c) must have a potency (potency(c)) one lower than its type's potency (potency(c_t)), except for star potency.*

$$\begin{aligned} PotencyCorrect(c, c_t) := c_t = type(c) \\ \wedge ((potency(c) \geq 0 \wedge potency(c) = potency(c_t) - 1) \\ \vee potency(c_t) = * \implies potency(c) \in \{*, \mathbb{N}_{\geq 0}\}) \end{aligned}$$

Potency correctness (Definition 9.2.1) is the most basic requirement for a deep model to be well-formed. The potency of a clabject defines the depth of its classification tree. Hence, a clabject with a potency of three has a classification tree of depth three meaning that direct and indirect instances exist in the next three classification levels. Following this rule all instances of a type must have a classification tree with a depth of one less than that of their type. Hence, their potency must be one lower than their type's. An exception to this rule is *star* potency (*), which represents clabjects with an unspecified classification tree depth. However, instances of a clabject with *star* potency can constrain their classification tree depth by defining a *non-star* potency or leave the depth of the classification tree unspecified by defining a *star* potency, too.

The attribute correctness well-formedness property (Definition 9.2.2) is satisfied if all clabjects which conform to their type in terms of potency possess at least the number of conforming attributes defined by their type. An instance containing the same number of attributes as its type is called an *isonym* while an instance containing more attributes is called a *hyponym* in [127]. An attribute conforms if its durability is not negative and one lower than that of the

corresponding attribute at the type level (or in the case of a *star* durability (*) is not negative or *star* itself), the data type and name are equal to the type level attribute and the value is the same as the value at the type level in cases where the type level attribute has a mutability of zero. Mutability has to be one lower than at the type level unless it is zero. If the mutability at the type level is the *star* value, it has to be non-negative or *star* itself. In addition, the mutability cannot be higher than the durability. If the mutability at the type level is zero it has to be zero at the instance level, too.

Definition 9.2.2 (Attribute Correctness)**.** *For every attribute (a_t) of clabject c's type (c_t) with a durability (durability(a_t)) greater than* 0*, the instance needs to have one conforming attribute (a_i).*

$$
\begin{aligned}
&AttributeCorrect(c, c_t) := c_t = type(c) \\
&\quad \wedge (\forall a_t \in attributes(c_t) : durability(a_t) > 0 \implies \exists a_i \in attributes(c) : \\
&\qquad name(a_i) = name(a_t) \\
&\qquad \wedge ((durability(a_i) \geq 0 \wedge durability(a_i) = durability(a_t) - 1) \\
&\qquad\quad \vee durability(a_t) = * \implies durability(a_i) \in \{*, \mathbb{N}_{\geq 0}\}) \\
&\qquad \wedge (mutability(a_i) = max[0, mutability(a_t) - 1] \\
&\qquad\quad \vee mutability(a_t) = * \implies mutability(a_i) \in \{*, \mathbb{N}_{\geq 0}\}) \\
&\qquad \wedge mutability(a_i) \leq durability(a_i) \\
&\qquad \wedge mutability(a_t) = 0 \implies value(a_i) = value(a_t) \\
&\qquad \wedge datatype(a_i) = datatype(a_t))
\end{aligned}
$$

Connection correctness (Definition 9.2.3) requires that each instance of a type must be connected to instances of connections which are connected to the type with a lower multiplicity of one (i.e. mandatory connections). These connections need to conform to their types in terms of connection conformance, i.e. follow rules for multiplicities, connection end names etc. as described in [20, 97]. Connection correctness includes clabjects which are connected to more connections than required by their type as it is the case with attribute correctness.

Definition 9.2.3 (Connection Correctness)**.** *For every mandatory connection (*$mandatory(con_t, c)$*) of a clabject c's type (*c_t*), the clabject needs to have conforming connections (*$conforms(con_t, con)$*).*

$$\begin{aligned} ConnectionCorrect(c, c_t) := & \, c_t = type(c) \\ & \wedge (\forall con_t \in connections(c_t) : mandatory(con_t, c) \implies \\ & \quad \exists con \in connections(c) : conforms(con_t, con)) \end{aligned}$$

For generalization correctness to be satisfied by instances they must fulfill the criteria defined in the inheritance relationship at the type level (*disjointness*, *completeness*). In contrast to violations of the classification correctness rules which can be detected without domain knowledge, violations of the generalization correctness rules are non-trivial to detect as domain knowledge is required for this task. Only modeled facts that contradict the implications of generalization relationships can be automatically identified. An example of such a contradiction would be an instance which is an instance of two clabject's participating in a generalization set marked as disjoint. Because of the limited possibilities to automatically judge violations of generalization correctness the following subsection focuses on emendation support for the violation of classification correctness rules.

9.3 The Emendation Service

According to [110] there are two ways to realize an emendation service — the difference-based and operation-based approach. The former creates model evolution operations based on a model difference analysis between the unchanged and the changed versions of the model. The latter requires a user to explicitly invoke model evolution actions while editing the model. Based on these user-invoked operations the model co-evolution operations are determined and recorded. A further refinement is the automatic recording of these operations as suggested in [128]. In both approaches, once the evolution operations have been identified they are applied to model instances conforming to the originally changed model.

There are three categories of changes for evolving a model [95]: 1. not breaking changes, 2. breaking and resolvable changes, and 3. breaking and

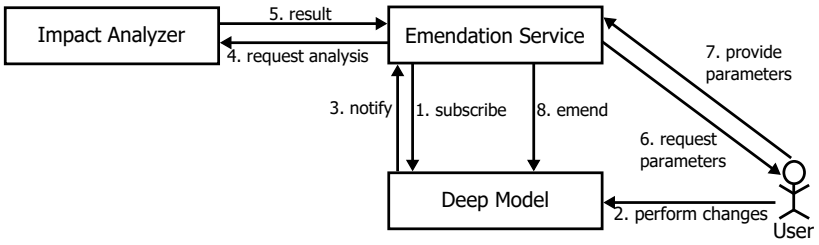


Figure 9.2: The emendation service architecture after [16].

unresolvable changes. Not breaking changes are changes which do not violate any classification rules, breaking and resolvable changes do break the classification rules but can be resolved automatically, while breaking and unresolvable changes can only be resolved with help from the modeler. The emendation service presented below focuses on breaking changes only.

The overall architecture of the emendation service is shown in Figure 9.2. It consists of three components — the Emendation Service, the Impact Analyzer and the Deep Model. The Emendation Service subscribes to changes to the Deep Model using the observer pattern. After a User makes a change to the Deep Model, the Emendation Service is notified. It then requests the Impact Analyzer to analyze whether the change made by the User violates *potency correctness* (Definition 9.2.1), *attribute correctness* (Definition 9.2.2) or *connection correctness* (Definition 9.2.3). The Emendation Service interprets this result and suggests emendation operations to the User that can be further parametrized. Finally, using the result of the Impact Analyzer and the parameters provided by the User, the Emendation Service applies emendation operations to the changed Deep Model.

As this architecture shows, this emendation service realization does not use calculated model differences between the base and changed models to calculate model evolution operations such as approaches described in [47, 84, 150, 229]. Instead, the emendation service directly supports the user while working with the model. A similar approach is suggested in [128] which is founded on the Praxis tool presented in [36]. The Praxis tool [36] records changes at runtime and is extended in [128] with an extensible change detection algorithm that can detect complex changes in the atomic evolution trace recorded using

the Praxis tool. The three most notable advantages of an approach based on operation recording are: 1. changes cannot hide each other (e.g. an attribute can be renamed before being moved which would lead to one delete and one add operation instead of a rename and move operation when applying a differencing algorithm), 2. the order in which the changes are executed is preserved and 3. atomic changes do not overlap (e.g. when two model elements are connected via a reference and a generalization, the move operation of an attribute to the superclass can overlap with the movement via the reference instead of the generalization relationship [128]). The emendation approach presented here essentially works in the same way as [128] and [36]. As the co-evolved models are not decoupled from the evolved model, emendation operations are performed after each atomic and complex change rather than collecting a trace of several changes and then calculating and executing the model evolution operations on a decoupled model instance as in [128] and [36].

9.3.1 Impact Analyzer

The emendation service and impact analyzer focus on the model manipulation operation categories identified in literature such as [110, 173, 186]. As a basis for the set of operations used for deep model emendation, the basic operations defined by Opdyke [186] are chosen. These are: 1. Creating an Entity, 2. Deleting an Entity, 3. Changing an Entity and 4. Moving an Entity. The first three are atomic changes to a metamodel whereas the fourth is a complex change composed of the create and delete operations. Other complex changes can be envisaged but are out of scope here. The following paragraphs provide an overview of how these operations occur in deep modeling and can violate the aforementioned classification rules.

Creating an Entity. In the case of deep modeling the operation for creating a program entity can be applied to entities, connections, features and inheritance relationships. The operations which affect classification correctness are the addition of attributes, connections and supertypes (through inheritance relationships) to entities and connections. The addition of an attribute effects *attribute correctness* (Definition 9.2.2) as instances of a clabject no longer have one conforming attribute for each attribute of their type. This is not a problem

for the clabjects in the level above the changed clabject (i.e. type level) because Definition 9.2.2 only restricts instances to having at least the attributes of their type without making any statement about the maximum number of attributes an instance can possess. Hence, instances can have more attributes than their type without violating the *attribute correctness* well-formedness rule. Nevertheless, an instance might change from an isonym to a hyponym. The add attribute operation is also indirectly executed when adding supertypes to a clabject through inheritance as the clabject inherits all attributes of the newly added supertype. When adding a connection to a clabject which is mandatory for instances (i.e. lower bound higher than one), *connection correctness* (Definition 9.2.3) is violated by instances.

Deleting an Entity. The *attribute correctness* constraint is violated by each delete operation that changes the set of attributes available to a clabject. Examples of such delete operations are the deletion of attributes, the deletion of clabjects serving as supertypes for other clabjects and the deletion of inheritance relationships. After such an operation has been performed, clabjects potentially no longer have all the attributes required to be a valid instance of their types. Moreover, *connection correctness* (Definition 9.2.3) is potentially violated by deleting mandatory connections or clabjects connected to other clabjects . However, it is not a problem to delete clabjects which are not associated to connections and do not participate in any inheritance relationship or classification relationships.

Changing an Entity. Changing an entity refers to changing the values of meta-attributes of clabjects and features. For *potency correctness* (Definition 9.2.1) the potency values of clabjects are relevant. Hence, changes to the potency of a clabject can effect the validity of classification relationships in the type and instance direction. *Attribute correctness* (Definition 9.2.2) is effected by changes to the name, data type, durability, mutability and values of an attribute. Again, this effects types as well as instances of the clabjects which contain the changed attribute and their subclasses. Also *connection correctness* (Definition 9.2.3) can be effected by changes to traits of connections and their connection ends such as lower and upper cardinalities. A connection

that was originally not mandatory, for instance, can become mandatory after a change to the cardinality of a connection.

Moving an Entity. The move operation is a composition of the delete operation at the source and the add operation at the target. Hence, it can be treated as a complex evolution operation with the effects of a remove operation at the source followed by an add operation at the target.

```
Data: changedClabject, changedTrait, oldValue, newValue
Result: List of by a change possibly impacted clabjects.
1 impact;
2 if classificationEffectingChange(changedClabject, changedTrait,
    oldValue, newValue) then
3   possibleImpact ← buildClassificationTree(changedClabject);
4   for current ∈ possibleImpact do
5     if violatesClassification(changedClabject, current, changedTrait,
        oldValue, newValue) then
6       impact ← impact ∪ current;
7   end
8 end
9 return impact;
```

Algorithm 9.1: The impact analyzer algorithm.

The operations presented previously and the checking of the rules they violate are implemented in the Impact Analyzer. The algorithm describing its operation is displayed in Algorithm 9.1. The algorithm expects the changedClabject, the changedTrait, the oldValue and newValue as input and calculates a list of all model elements which are impacted by the change. This list not only contains the minimal set of changes needed to keep classifications valid but is extended to include the maximum set of all impacted model elements. When adding an attribute, for example, it would be possible to calculate the impact only in the instance direction of the classification tree. For a modeler, however, it can also be desirable to additionally add the attribute in the type direction to apply a more strict style of *attribute correctness* (Definition 9.2.2) which

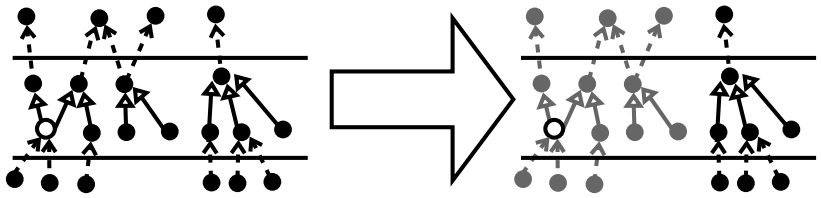


Figure 9.3: The impact of a change.

would ensure that instances do not have more attributes than their types (i.e. are isonyms only).

The algorithm first checks in line 2 whether the change is one of the former described changes which effect either *potency correctness* (Definition 9.2.1), *attribute correctness* (Definition 9.2.2) or *connection correctness* (Definition 9.2.3). Examples of such changes are a change to the potency of a clabject or the addition/removal of features from a clabject. If a change potentially violates the classification rules, the algorithm builds up the classification tree for the changed model element using the buildClassificationTree(changedClabject) operation in line 3. The classification tree is the transitive closure over all classification and generalization relations which originate from the changedClabject. For each of the elements in the classification tree the algorithm then checks whether one of the definitions for classification correctness (Definition 9.2.1 - Definition 9.2.3) is violated by the change (line 5). If the change to the clabject violates the classification rules it is included as an impacted item (line 6). The algorithm terminates after the whole classification tree of the changedClabject has been checked and returns the impact in line 9.

Figure 9.3 illustrates a change and the result of a run of the impact analyzer presented in Algorithm 9.1. The figure shows two abstract deep models consisting of three levels in which clabjects are represented by black solid circles connected via classification relationships (dashed arrows) and generalization relationships (solid arrows). The left-hand side shows the model with a change applied to a clabject indicated by a non-solid circle. The impact calculated for the change is shown on the right-hand side of the figure, indicated in gray. It can be observed that a change to one model can have multiple knock-on changes which makes it important to help users keep track of things. The

emendation service supporting these changes is presented in the following section.

9.3.2 Emendation Service

Depending on the kind of change made to a deep model, different emendation operations are suggested to keep classification relationships valid. These are listed in the following paragraphs:

Add/Remove/Move Attribute. The move attribute operation can be subsumed by the add and remove operations on attributes, because the move operation can be understood, as previously described, as a remove operation at the source followed by an add operation at the target. Both the add and remove operations on attributes have an impact on the classification relationships in a deep model. The minimum change which has to be performed after adding an attribute to a clabject is to add the same attribute to the instances of the changed clabject. The instances then conform to their type again because they have at least the set of attributes required by their type (Definition 9.2.2). As an instance can have more features than the type it is left to the modeler to configure an emendation service in such a way that the types and their instances also obtain new attributes that are added. When removing an attribute which is defined at the changed clabject's types, it also has to be removed in the type direction of the classification hierarchy to retain *attribute correctness* (Definition 9.2.2). It is left to the modeler to configure the emendation service so that the feature is also removed in the instance direction. This is not required to satisfy the *attribute correctness* well-formedness property because instances are allowed to posses more attributes than their types.

Add/Delete Connection. Connecting and disconnecting connections to and from clabjects can effect *connection correctness* (Definition 9.2.3). *Connection correctness* requires all instances of a clabject to have conforming connections for type level connections which are defined as mandatory by their lower cardinality constraint. All connections, both mandatory and not mandatory, must adhere to the cardinality constraints at the type level. In the case

of the addition of a newly created connection, the user is not supported by the emendation service since all connections are initially connected to clabjects with a lower multiplicity of zero allowing instances of the connected clabject to be valid without having instances of this connection connected. The delete operation, however, can affect a clabject's conformance to its type and should, thus, be supported by an emendation service.

Add/Remove Supertype. Adding and removing supertypes of a clabject in an inheritance hierarchy results in a change to the set of attributes possessed by the clabject with the added or removed supertype in its inheritance hierarchy. In the case of the addition of a new supertype, the emendation operation for adding an attribute has to be executed for each newly inherited attribute and in the case of the removal of the supertype the remove attribute operation has to be executed for each attribute which is no longer inherited. The same applies to inherited connections.

Change Attribute Traits. The events which effect classification relationships when traits of features are changed are changes to the durability, mutability, data type, name and value of an attribute as indicated by Definition 9.2.2 describing *attribute correctness*. When a change is made to the durability of an attribute, all model elements in the classification tree are analyzed to determine whether the feature needs to be added or removed at instance levels. Additionally, the durability value has to be recalculated for the whole classification hierarchy. Mutability has an effect on the values of attributes. When a change is made to the mutability of an attribute it has to be determined whether the value of any instance's attributes has to be set to the value defined by its type. Like durability, the analysis has to be performed on the whole classification hierarchy. A change to the data type and name of an attribute requires this change to be made to all derived attributes in the classification hierarchy. When changing the value of an attribute with mutability *zero*, the new value has to be propagated to the effected attributes.

Change Clabject Traits. Changes to the traits of a clabject as well as to the connections they are connected to can effect the validity of classifica-

tion relationships. The only trait of clabjects which influences classification is the potency trait as defined in the *potency correctness* (Definition 9.2.1) well-formedness rule. A change to this trait immediately effects the maximum depth of the classification tree that can be generated from the clabject. If the potency value is reduced the depth of the classification tree has to be decreased, either by deleting classification relationships, which creates untyped clabjects, or by deleting clabjects which would otherwise have a negative potency (which is forbidden). Potency is recalculated throughout the whole classification tree after a potency change no matter whether it is increased or decreased. Changes to traits of connections and their connection ends (e.g. lower cardinality bound) effect *connection correctness* (Definition 9.2.3). Hence, emendation operations have to be offered to ensure that a clabject conforms to its types after changes to such traits. These include 1. automatic addition and deletion of connections, 2. automatic renaming of connection end monikers, and 3. automatic adjustment of connection multiplicities.

```
   Data: changedClabject, changedTrait, oldValue, newValue, operation
   Result: All instances are updated to conform after the operation
 1 impact ← calculateImpact(changedClabject, changedTrait, oldValue,
     newValue);
 2 if impact ≠ ∅ then
 3 |   options ← queryOptionsFromUser(changedClabject, changedTrait,
   |     oldValue, newValue, operation, impact);
 4 |   impact ← reduceImpactBasedOnOptions(changedClabject,
   |     changedTrait, oldValue, newValue, impact, options);
 5 |   if impact ≠ ∅ then
 6 |   |   for current ∈ impact do
 7 |   |   |   applyEmendationOperation(current, changedClabject,
   |   |   |     changedTrait, oldValue, newValue, options);
 8 |   |   end
 9 |   end
10 end
```

Algorithm 9.2: The emendation service algorithm.

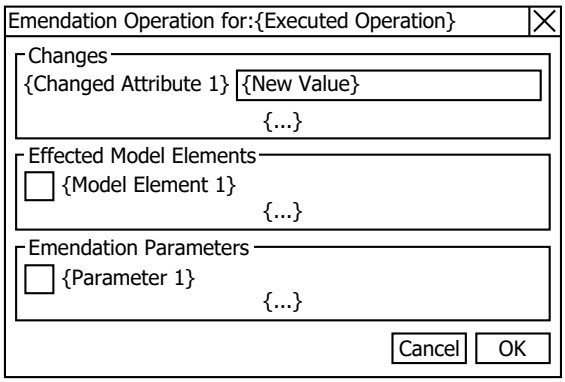


Figure 9.4: A dialog for querying emendation parameters.

The algorithm applied by the emendation service is shown in Algorithm 9.2. The expected input to the algorithm is the changedClabject, changedTrait, oldValue, newValue and executed operation. The effect of the algorithm is to change all model elements in changedClabject's classification tree to ensure that all classification relationships are valid after a change. The emendation service first uses the previously introduced impact analyzer algorithm to calculate the impact of a change (line 1). As earlier mentioned, this impact is calculated in such a way that the entire set of clabjects that are possibly effected by a change are taken into account.

If the impact analyzer identifies an impact on the model (line 2), the user is queried about the options that the emendation operation could apply (line 3). These options, together with the possible parametrization of the emendation service, allow modelers to use the emendation service in a way that best fits their modeling style. For example, a user can apply a style in which types and instances must always have exactly the same number of attributes or in which attributes are only added to supertypes and never to subtypes.

Figure 9.4 shows a mockup of a dialog box querying the user for such a parametrization decision. The title of the dialog shows the identified operation ({Executed Operation}), e.g. *Change Potency* or *Add Attribute*. The Changes group shows the applied change, e.g. a change to the durability of an attribute. The Effected Model Elements group displays model elements effected by the change (i.e. the impact). By deselecting model elements, a user can

exclude them from consideration. The Emendation Parameters group allows the emendation algorithm to be further configured to fit a modelers style of modeling. For example, a modeler could choose to add an attribute to instances only and not to types, or to add an attribute to supertypes of instances only.

Based on the parameters provided by the user, the impact is recalculated (line 4) and the set of model elements to emend is potentially reduced. If the recalculated set of impacted model elements is not empty, these elements are emended (line 7). The emendation service applies the previously introduced emendation principles during emendation.

9.4 Limitations

Even though the emendation service described above is fully functional and is useful, there are many opportunities for improvements and new functionality. For example, it is possible to support dynamic definitions of classification consistency and emendation rules. Dynamically changing the consistency rules without the need for tool re-deployment allows different styles of deep modeling to be created with relaxed or more strict classification rules as desired by a modeler. Also, the addition of emendation rules on-the-fly allows them to be adapted to specific modeling styles dynamically. To realize such a feature the emendation service architecture can be enhanced as shown in Figure 9.5. This enhancement proposes an additional Consistency Requirements definition containing consistency rules as input to the Impact Analyzer and Emendation Operations definitions as input to the Emendation Service. The emendation operations can then be mapped to violated consistency requirements and thus automatically selected and displayed to the user for selection. A version of an impact analyzer running on a recorded change trace using this feature is suggested in [128] for example. Their so-called Complex Change Detection Engine takes definitions of changes as input so that it can detect new kinds of changes on-the-fly.

The proposed parametrization of the emendation service can be enhanced to allow different decisions to be taken on each subtree of the classification tree rather than on the whole classification tree as currently suggested. To realize this, the suggested emendation parametrization UI mock-up displayed

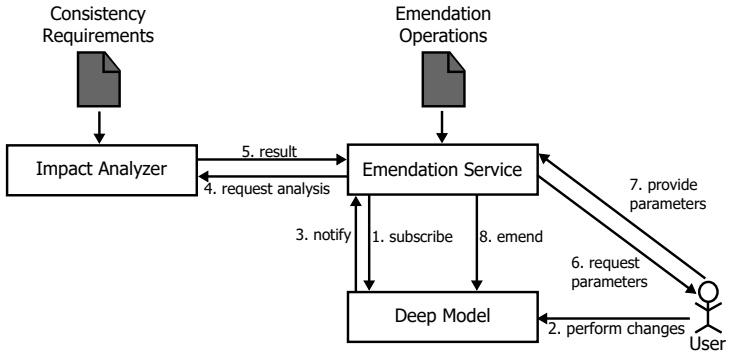


Figure 9.5: The extended emendation service architecture after [16].

in Figure 9.4 needs to visualize the classification tree, allow subtree selection and accept corresponding emendation parameters for each subtree. The emendation algorithm as presented in Figure 9.2 would also have to be adopted to focus on subtrees rather than always on the whole classification tree.

Currently the emendation service focuses on applying changes to models which are stored locally. However, it is possible to build a web of deep models which reference each other. The presented approach in this chapter does not scale to such a scenario since local changes can also logically effect remote model content linking to the changed model. To support such a case, change recording technologies can be applied which record local changes and transport them to remote models containing links to the changed model on request. Research on this has for example been done in the ontology evolution domain in [132] and modeling in [128]. Such approaches can be adapted to support emendation and evolution in linked deep-model scenarios.

Another possible extension of the emendation service is to support clabject retyping as suggested by de Lara et al. in [56]. This deep modeling approach allows the types of clabjects to be changed dynamically during run-time. However, this is not covered by the emendation service described in this chapter since it focuses on the definition and usage of deep user-defined languages and therefore assumes a constructive rather than exploratory modeling approach [28]. In constructive modeling, instances are built from their types, so the types of clabjects rarely change after creation. In exploratory modeling, however, a bottom up approach is applied which builds types from instances,

so possible types of instances are discovered after the instances have already been created. If a user retypes a clabject it has to be manually ensured that the classification conformance rules are adhered to in the current version of the emendation service. In the tool described in [56] this is taken care of automatically.

In [96], de Lara et. al describe an approach for multi-level model-satisfiability checking and model completion. This approach can be used to check that a model satisfies existing constraints after a change or to generate new model elements to ensure that a model adheres to newly defined constraints.

When a model evolves, the artifacts accompanying it also have to be evolved. These include constraints, transformations, interpreters, concrete syntax etc. This, however, is beyond the scope of the emendation service presented here but is an option for future research. A tool supporting the evolution of constraints in a deep modeling framework is Cross Layer Modeler [57]. The co-evolution of transformations is supported by the CO-URE tool presented in [85], while Di Ruscio et al. [58] support abstract and concrete syntax co-evolution.

Chapter 10

Deep, Seamless, Multi-format, Multi-notation Modeling with Melanee

In [26] the idea for a deep modeling environment was described for the first time by Atkinson et al. This represented the start of a quest to develop a deep modeling environment at the University of Mannheim which ended in the deep modeling environment Melanee [8] developed as part of this thesis. This section first describes the architecture of Melanee and then walks through an example of how to create a language that demonstrates the key features of the tool.

10.1 Melanee Architecture

Melanee is built using the Eclipse Platform as indicated by Figure 10.1. This provides a stable, well-tested foundation with long maintenance cycles. The specific features of Melanee are realized by leveraging state-of-the-art modeling technology to the greatest extent possible. The metamodel, which is based on

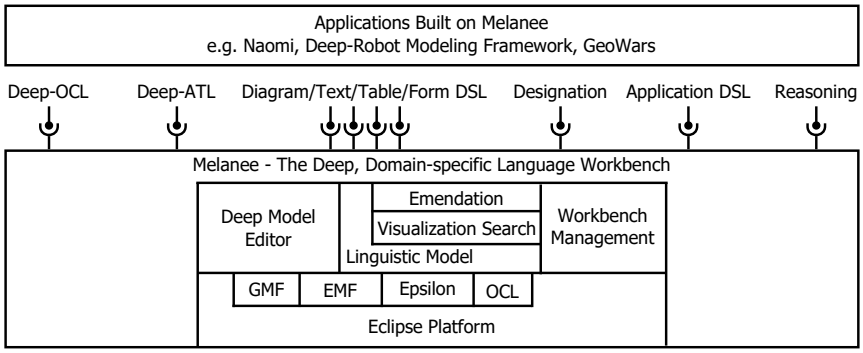


Figure 10.1: The Melanee architecture overview.

the definitions presented in [127] is implemented using EMF modeling technology [213], a de-facto industry standard for model-driven development at the time of writing. The query operations provided by the metamodel elements are implemented using OCL body statements [177, 184], while constraints on the metamodel are defined and validated using the Epsilon Validation Language [139] of the Eclipse Epsilon Framework [187]. This language basically resembles the OCL [184] when it comes to defining constraints but allows them to be enriched with richer information for displaying error messages and offering automatic *quick fixes*. The diagrammatic Deep-model Editor is implemented using the Graphical Modeling Framework [94]. Wherever features of the model editor are not provided by the default GMF modeling capabilities, they are added by extending its XPand-based [69] code generation templates using aspect-oriented technology [197]. The combination of these technologies made it possible to implement Melanee almost exclusively with well-established, model-driven technologies.

The core component shipped with the basic version of Melanee is the *Melanee Workbench* which offers extension points and extension point management features for all functionality except the hardwired diagrammatic predefined language editor. All other parts of Melanee can be added and exchanged to fit a particular user's needs as indicated by the extension points in Figure 10.1. Default implementations of various extensions are available for installation via Melanee's plug-in manager — namely, 1. a deep constraint language (Deep-OCL), 2. a deep, rule-based transformation language (Deep-ATL), 3. dia-

grammatic, pre and user-defined languages (Diagram DSL), 4. textual, pre and user-defined languages (Text DSL), 5. tabular pre and user-defined language (Table DSL), 6. form-based pre and user-defined languages (Form DSL), 7. a service for deriving names of modeling elements (Designation), 8. a language to model the configuration of the modeling environment itself (Application DSL) and 9. a service for executing ontology like reasoning operations on a deep-model (Reasoning). Custom applications are built on top of all these plug-ins and the Melanee workbench. Examples are the deep orthographic modeling environment Naomi [24], the Deep-Robot Modeling Framework [21] and the GeoWars game [22].

The metamodel on which all plug-ins are build, the PLM, is shown in Figure 10.2. The root class for all meta types is Element which defines the name attribute for all types and connects them to their LMLVisualizer. The LMLVisualizer stores the visualization parameters for model elements when visualized in the predefined, diagrammatic LML. These parameters are the location relative to the container (xLocation and yLocation), the size (width and height) and an extensible list of rendering parameters in attributes. This list specifies how each linguistic trait of a model element is visualized. The options range from *default*, which applies the default information hiding rules of the LML, over *noshow*, which always hides the trait, and tvs, which displays the trait value in the trait value specification below a clabject's name, to *show* which ensures that the trait is always visible regardless of the LML elision rules. Additionally, for each format the list stores what notation a model element is visualized in. At the time of writing *text*, *diagram*, *form*, *table* and *app* are available. The application (app) format is not a format in the usual sense, rather, its visualizers are used to configure the modeling workbench in which the format-specific editors are displayed. Using the feature it is, for example, possible to configure the menus, views and toolbars visible in the modeling workbench. The list of LMLVisualizer attributes can, however, be extended as necessary.

User-defined visualization of model elements is defined through the AbstractUserDefinedVisualizers attached to the LMLVisualizer. As many AbstractUserDefinedVisualizers for as many formats and notations as needed can be attached to an LMLVisualizer. For each format, there is a subclass of AbstractUserDefinedVisualizer which contains a format-specific notation description. The no-

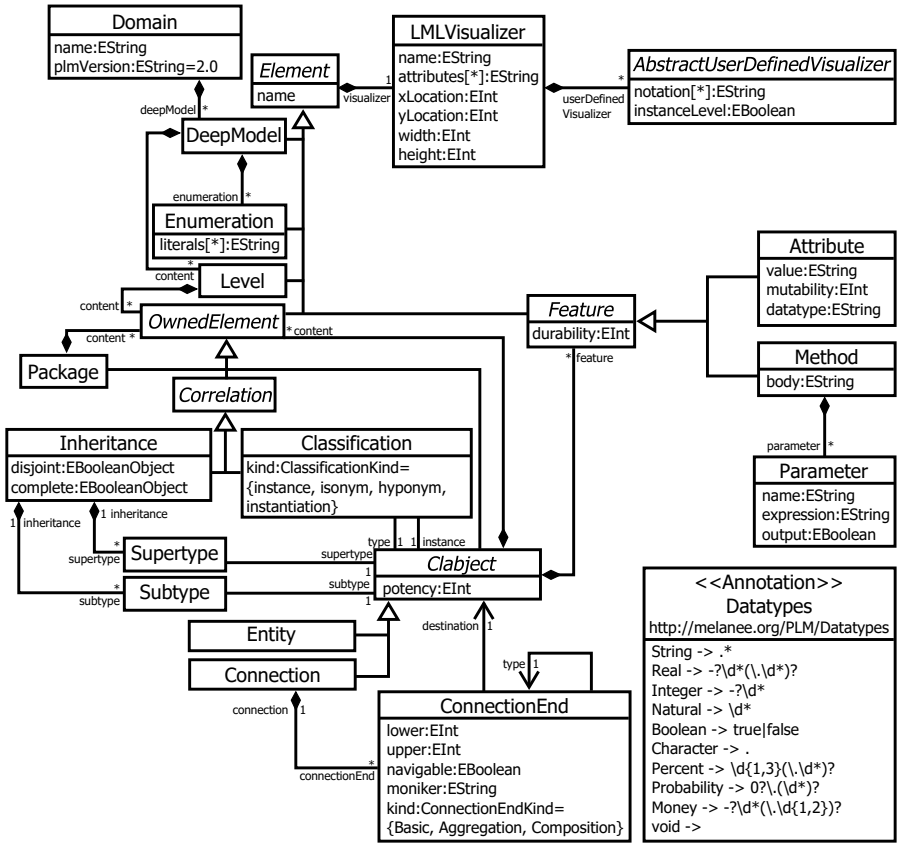


Figure 10.2: The in Melanee implemented PLM.

tations to which an AbstractUserDefinedVisualizer contributes are referred to in the notations list. Furthermore, it is possible to configure whether to apply an AbstractUserDefinedVisualizer to the instanceLevels only or in addition to the level in which the visualizer containing model element resides (i.e. the type level).

The Domain class is the outer most container for DeepModels. It has a name identifying the domain and an attribute storing the version of the employed linguistic metamodel (plmVersion). DeepModels store the Levels which organize domain content into ontological classification levels. All level content that is contained in a container inherits from the OwnedElement type. These OwnedElements are either Packages, Correlations or Clabjects. The Inheritance and Classification Correlations describe logical relationships between the domain concepts

(i.e. Connections and Entities). The Inheritance relationship specifies whether one class is a subclass of another. Superclasses are connected via the supertype reference and subclasses via the subtype reference. An Inheritance relationship can connect an infinite number of Subtypes and Supertypes enabling multiple inheritance. Furthermore, an Inheritance can be characterized using the disjoint and complete attributes to make set theoretic statements about instances of subclasses. Distinct classes were chosen for Supertype and Subtype to make mappings to relational databases more natural since tables storing the connection information are explicitly created for super- and subclasses. The Classification relationship expresses the fact that one Clabject is an instance of another. Classifications are restricted to connecting Clabjects at adjacent levels and are always stored in the container of the instance end. Four kinds of Classifications, specified through the kind attribute, exist: instance, isonym, hyponym and instantiation. Instance classifications do not specify further what kind of instances are connected to the classification. Isonym and hyponym declare whether the instance has exactly the same attributes as the ontological type (isonym) or whether the instance has additional attributes beyond the minimal set needed for conformance to the type (hyponym). Instantiation declares that an instance has been created through instantiation rather than, for example, through type inference by a reasoning service.

Domain content is expressed through subclasses of Clabject and their characterizing Features. Two subclasses of Clabject exist which are Entities, representing entities in the problem domain, and Connections, describing relationships between Entities in the problem domain. A Clabject's potency is defined by the potency trait. In the deep modeling approach supported by Melanee Entities and Connections can be further characterized through Features. Thus, all Connections are equivalent to association classes in the UML. Connections are connected to Entities through ConnectionEnds which are stored in the Connections. ConnectionEnds are not first class model elements themselves, they exist to group ConnectionEnd traits so that Connections can be implemented more efficiently. A ConnectionEnd specifies: 1. the moniker used to refer to a ConnectionEnd, 2. whether the connection end is navigable , 3. lower and upper cardinality constraints and 4. a kind (basic, aggregation, composition). A basic ConnectionEnd does not convey any kind of containment or ownership relation-

ship. Aggregation requires aggregated instances to be connected to a connection with its connection end kind set to aggregation or composition. The aggregation kind, however, places no restriction on where the instances are physically located. Composition, the most restrictive kind, forces the instances to be owned by an instance of the composition connection end and to connect to it through a Connection of kind composition.

Features further characterize Clabjects. Two flavors of features exist as subclasses of Feature — Attribute and Method. Both have a durability trait describing their durability. Furthermore, Attributes have a mutability trait to define the mutability of an Attribute's value. In addition to the potencies, Attributes have a value trait describing the current value of the attribute and a data type describing the data type of the value. Data types are predefined in the PLM through the Datatypes annotation attached to the EMF package containing the metamodel. The annotation defines the available data types through tuples mapping the data type name to a regular expression in order to check the value trait of Attributes for correctness regarding their data type. Methods have a body trait which defines the functionality of the method in a constraint or action language. A method's inputs and outputs are represented by Parameters. Each Parameter has a name trait to identify the Parameter in the body of the owning Method. The expression trait specifies the data type in a constraint or action language's syntax. By setting the output trait to *true*, a Parameter can be declared as the result of a method.

All potencies — clabject potency, durability, mutability – and the lower and upper bounds of ConnectionEnds are of data type EInt which is an integer data type. To present the infinite **-value*, *-1* is used as in other modeling frameworks such as the EMF.

10.2 Melanee Tool Walkthrough: Creating a Domain-specific Language for Enterprise Modeling

This walkthrough shows how to build a part of a language for enterprise architecture modeling in Melanee. The language focuses on features needed to

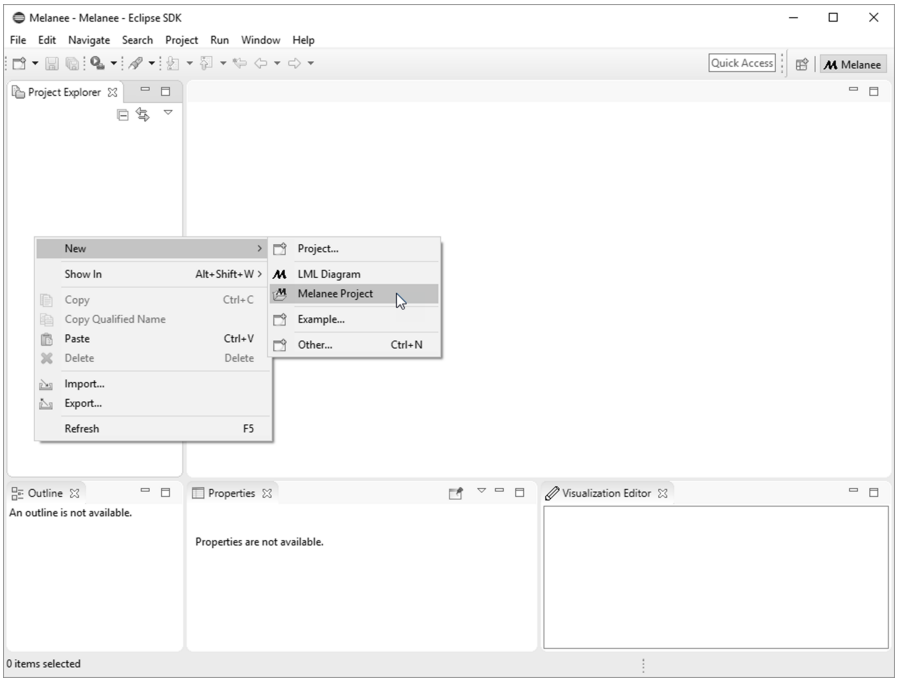


Figure 10.3: Melanee after start with the context menu for project and file creation opened.

model the structure of companies, departments and their employees. In ArchiMate this language would be regarded as supporting the description of part of the active structure of the business layer. Four formats are made available for editing — namely, graphical diagrams for communicating the company structure, text for rapid content creation by a technical user, tables for providing a condensed and efficient overview and forms for data administration by business focused employees.

Since Melanee is an Eclipse-based product it uses the concepts of workspaces, projects and files to organize information. The workspace, which is chosen while starting Melanee, is the place where all information managed by Melanee is located. Projects are then created in the workspace to organize files. Figure 10.3 shows Melanee just after it has been started with the plug-ins for reasoning and diagrammatic, textual, tabular and form-based languages installed. The Melanee perspective is displayed as indicated by the Melanee

text in the tool bar located in the upper right of Melanee. On the left the Project Explorer is shown with the context menu invoked to provide menu items for creating Melanee Projects and LML Diagrams. The view below the Project Explorer shows an outline indicating which excerpt of the model is currently being viewed in the model editor usually located at the center of the modeling environment. The bottom left shows the Properties view which allows the properties of any selected element to be changed. Finally, located to the right of the Properties view is the Visualization Editor which can be used to define the visualization of model elements in all formats supported by Melanee.

Once a project has been created, which is named Quality Toys in the example in Figure 10.4, files containing models can be added. Here, a file called Company Structure.lml is created containing the company structure of the company to be modeled. The model in the figure shows a first draft of the company structure language. A deep model named Quality Toys Structure has been created containing two levels — O_0 and O_1. The highest level is populated with the types present in the domain to be modeled. These are CompanyType, DepartmentType and EmployeeType. The connections between the model elements indicate that a CompanyType is composed of an unlimited number of DepartmentTypes which itself is composed of an unlimited number of EmployeeTypes. These concepts are in the process of being instantiated at level O_1. At the time when the snapshot was taken, a ToyCompany instantiates CompanyType, and DepartmentType is instantiated by MarketingDepartment and ResearchDepartment. A ToyCompany consists of one MarketingDepartment and one ResearchDepartment as indicated by the instances of the connection between CompanyType and DepartmentType.

Model elements are instantiated by selecting the type to instantiate from the Palette located on the right side of the deep model editor and clicking at the location where the model element is to be placed. While selecting a location for the model element to be instantiated, the user is given visual feedback on whether the instantiation is allowed at the current location. If instantiation is allowed at the current position the mouse cursor displays a small plus sign, otherwise it displays a red strike-through circle. The elements to be instantiated are grouped into groups for DeepModel Definition, Domain Definition, Correlation Definition and DSL Elements. The DeepModel Definition group contains Deep Model and Level since these are the outer most containers from which

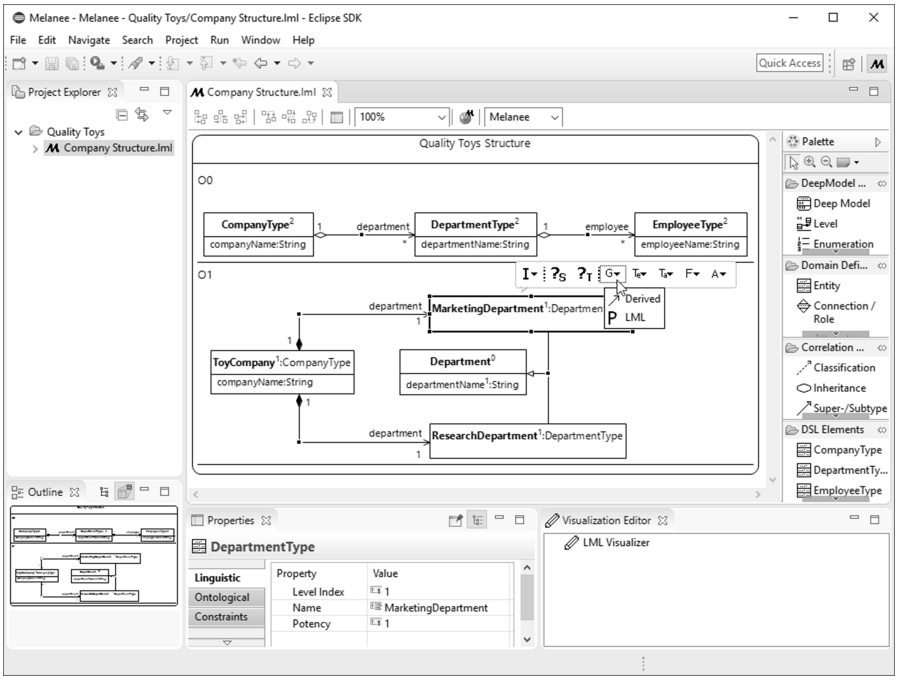


Figure 10.4: Melanee while building up the company structure modeling language.

every deep model is composed. Enumeration is also contained in this group since in the LML enumerations are owned by deep models and can be used at all classification levels. The second group, Domain Definition, contains all types which are needed to define domain content. These are Entity, Connection/ Role, Attribute and Method but not all of these are shown in Figure 10.4 for space reasons. Logical relations between domain elements are expressed through correlations located in the Correlation Definition group, which contains Classification and Inheritance with its Super-/Suptype endings. The instantiation of ontological types is supported through the DSL Elements group. This group allows ontological types to be instantiated in a context-sensitive way based on the model element currently selected in the diagrammatic editor. In the example, a model element at O_1 is selected (MarketingDepartment) so the palette offers the ontological types CompanyType, DepartmentType and EmployeeType for instantiation at level O_1.

The properties view at the bottom allows the model element selected in the diagrammatic editor to be modified. The tab pages on the left of the view allow the model element to be edited either from the linguistic point of view using the Linguistic tab page or the ontological point of view using the ontological tab page. The Constraints tab page allows constraints in a deep constraint language to be defined on the selected model element. Additionally, a *Visualization* tab page is available for manipulating the *LML* visualization of a model element but is not shown for space reasons. Here, the linguistic properties of the selected MarketingDepartment model element are shown in the properties view. Name and Potency of the MarketingDepartment can be set. The Level Index entry is for informational purpose only and helps with orientation on big deep models where the level of a model element is not immediately obvious.

Above the selected MarketingDepartment model element a toolbar is displayed. This toolbar is extendable by plugins available with Melanee and offers quick access to the most important actions available on the selected model element. The first icon — I — is used to select the model element for instantiation at a lower level. The arrow pointing to the bottom next to the I indicates that more than one option for instantiation is available. These are instantiating the model element without content and instantiating the model element and its content together in one step. The following two tools — $?_S$, $?_T$ — set the current model element as the source or target element for the reasoning service. The next four tools — G, T_e, T_a, F — set the format and notation to be used for the selected model element, in this case the graphical/diagrammatic (G), textual (T_e), tabular (T_a) and form-based (F) formats. The drop-down values of the selection shows the notation to display. Since no user-defined notations have been created at the time the snapshot was taken, only Derived and LML are offered. Selecting Derived causes the model element to use the notation selected to visualize its parent container, which is the level O_1 in the case of MarketingDepartment selected here, whereas the selection of LML uses the predefined LML language for visualization. The last tool — A — switches the modeling environment into a layout defined through application visualizers. Again, different notations (i.e. layouts) are provided by the drop down list of the tool.

Located at the top of the editor are tools related to the editor as whole and not to a single, selected model element. The first two groups of tools help align multiple selected model elements with options like left-align, center-align etc. This group is followed by a group with a single icon which opens the properties view if it is closed. The next group contains a drop-down box which offers different zoom levels for the diagram followed by a tool group which contains a button to open a dialog for discovering remote deep models and linking content from these remote models into the currently edited deep model. Finally, a selection is available to select whether the model shall be validated against consistency rules of the Melanee or MetaDepth approach.

Once the domain model has been created it can be enriched by making further changes. In the current state of the example only the name attributes (companyName, departmentName, employeeName) have been modeled. When new information is added, the consistency of the model can be maintained using the emendation service which automatically detects changes to the model and offers options to keep the classification relationships valid. When adding the name attributes to CompanyType, DepartmentType and EmployeeType in Figure 10.4 the emendation service ensured that these attributes were also added to existing instances as necessary

In Figure 10.5 the salary attribute is currently being added to EmployeeType while the snapshot is taken. This change is detected by the emendation service which calculates that MarketingExpert and Researcher need to be emended, i.e. also need the salary attribute. The user is notified of this need by the emendation service through the dialog shown in the bottom left of the figure. The dialog queries the user for the traits of the attribute to be added — Name, Datatype, Durability and Mutability. Further configuration of the emendation service is possible through the checkboxes at the bottom of the dialog. These options include whether duplicates should be prevented by the emendation operation or whether certain directions in the classification hierarchy shall not be considered by the emendation service. The emendation service can be switched off globally by deselecting the tick in the Enable emendation service checkbox, and can be re-enabled in the global Melanee preferences dialog. By pressing OK the operation is executed on the whole model, while by pressing Cancel the operation is only executed for the changed model element. In the example the

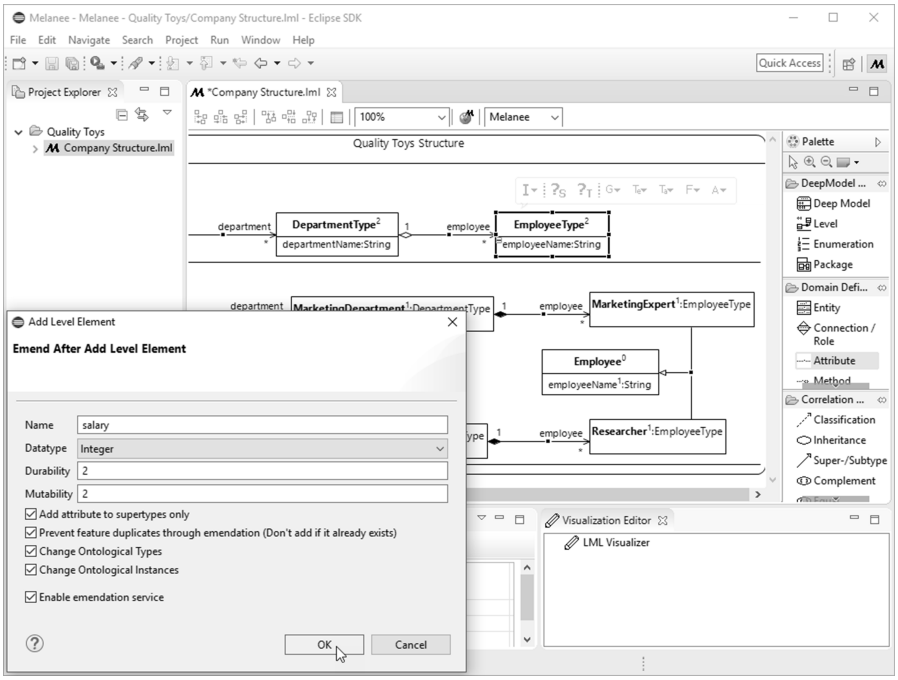


Figure 10.5: Configuration of the emendation service to add the salary attribute to EmployeeType.

Add attributes to supertypes only option is checked so that the attribute is added only to Employee, the supertype of MarketingExpert and Researcher. Furthermore the emendation service is configured to execute changes in both the type and instance directions and prevent duplicated features where necessary.

After completing the model with all attributes and types, user-defined notations can be defined. In this case a diagrammatic notation for communicating the company structure, a textual syntax for fast model content creation, a tabular syntax for analyzing data and a form-based syntax for daily work tasks are created. The graphical syntax represents instances of CompanyType as a house, instances of DepartmentType as boxes dividing the house and the EmployeeTypes as a group of stickmen. The visualization of EmployeeType is further refined by MarketingExpert and Researcher which define a stickman with an M at its top right-hand side to represent the former and a stickman with an R at its top right-hand side to represent the latter. The textual syntax represents compa-

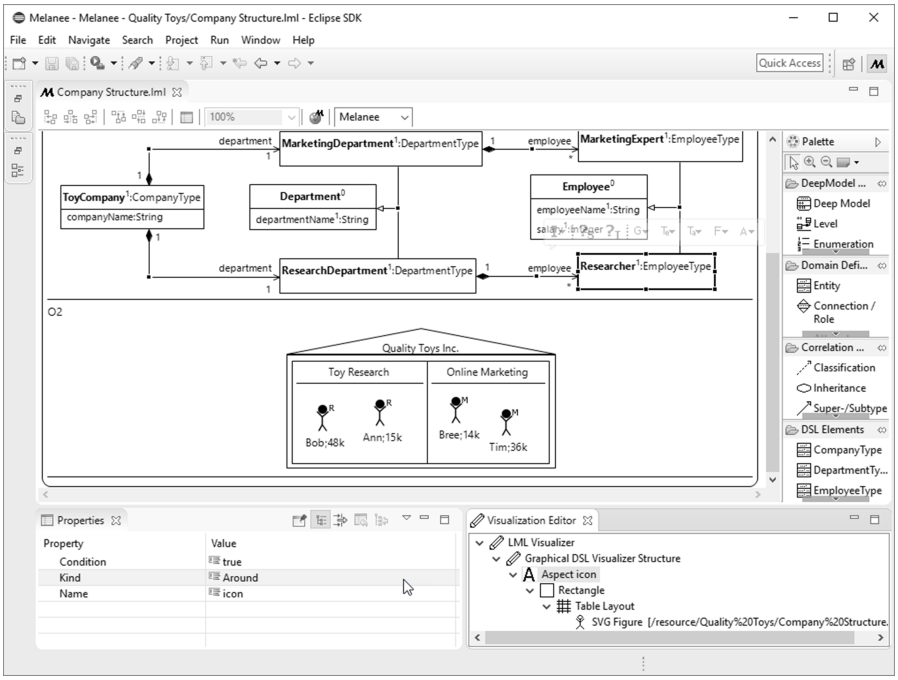


Figure 10.6: Diagrammatic user-defined language definition for Researcher using aspect-oriented features.

nies and departments through the keyword *company* and *department* followed by their name and their content in curly brackets. CompanyTypes contain DepartmentTypes and DepartmentTypes contain EmployeeTypes. EmployeeTypes are represented by their name followed by their attributes (salary) in brackets and terminated by a semicolon. The form-based syntax allows all attributes to be edited through text boxes and uses list boxes to add, remove and navigate to the content of CompanyTypes and DepartmentTypes. In the tabular format the connections and attributes of a model element are visualized through columns which can also be used for navigation.

The example in Figure 10.6 shows the definition of the user-defined visualization of Researcher. To gain more space for the deep model editor, the project explorer and outline are collapsed. The visualization is defined by adding a Graphical DSL Visualizer to the LML Visualizer of the model element in the Visualization Editor at the bottom right of the screen. The visualizer

is named after the notation, here Structure. Multiple visualizers of the same format can be defined but need to have different notation names. A user can later select one of the defined notations in which a model element is visualized from the format drop-downs in the toolbar (G, T_e, T_a, F). In the example of Figure 10.6 the user-defined language definition for Researcher is shown in the Visualization Editor. The shape itself is defined in the visualizer defined on Employee (not shown here) and Researcher provides an aspect only to modify this shape. The visualizer of EmployeeType consists of two vertically stacked (layout) rectangles not displaying their border. The top rectangle contains the stickman pictogram and is named icon while the lower rectangle contains the information about the employee (here employeeName and salary). The Researcher for which the visualizer is shown in the Visualization Editor customizes the icon of the user-defined symbol. The stickman is replaced by a stickman with an R at its upper right-hand side to depict the research nature of the employee.

Customizations of visualizations by ontological types or supertypes are defined by means of Aspects. Around is selected as the kind of the aspect resulting in the replacement of the definition called icon in the merged graphical visualizer. The provided condition is true, which applies the aspect to all instances of Researcher. The aspect consists of the Rectangle defining the upper part of the user-defined diagrammatic visualization. This rectangle contains a Table Layout with one column. In this, a stickman figure with an R at its upper right-hand side is placed. Defining notations in other formats can also be achieved using the tree-based Visualization Editor. For this purpose form, table and text visualizers can be added as children to the LML Visualizer.

Once all the shapes have been defined, the user can start to model with the user-defined language in Figure 10.6. In the example, the whole O_2 level has been switched to the Structure notation. It is also possible to switch individual model elements to a specific notation, but this is not shown in this example.

The resulting workbench after defining and invoking diagrammatic, textual, tabular and form-based user-defined languages on the company structure modeling language is displayed in Figure 10.7. The workbench includes four format-specific editors, a diagrammatic editor shown on the left of the screen (the palette located to the right of the editor is collapsed for space reasons), and a form-based, a table-based and a text-based editor stacked vertically on

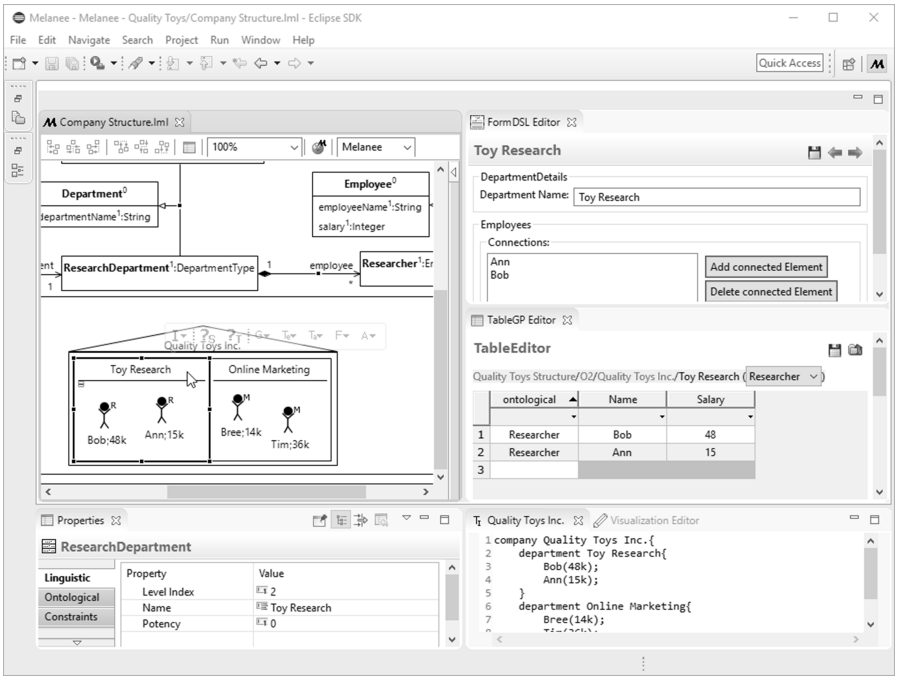


Figure 10.7: The Toy Research department edited in diagrammatic, form-based, tabular and textual user-defined languages.

the right-hand side of the screen. The department Toy Research is selected in all four editors and can be updated equally in any of them. The selection is indicated in the diagrammatic editor by a square surrounding the Toy Research shape, in the form editor by showing the department name in the form title, in the tabular format through its breadcrumb and in the textual editor by placing the cursor at the selected element (not shown here because the editor is inactive).

Usually a modeler would not interact with all four formats at the same time but only in a customized environment that offers the options needed for the task in hand. Such a customized environment can be defined using an application visualizer for the container of the model excerpt to be edited. In this case it would be defined for the whole level since it is intended that all level content is editable. The visualizer can be activated through the toolbar of the model element it is defined at. After activation, the environment switches

to the layout defined in the application visualizer. The application visualizer creator can define which views, menus, toolbars and pages in the Properties View are visible to the model user. Additionally, it is possible to define what format and notation the model content is to be edited in when activating an application visualizer. In the model shown here four application visualizers can be envisaged, one for the diagrammatic format and notation, one for the textual format and notation, one for the tabular format and notation and one for the form-based format and notation. All four show the Project Explorer but hide all other views. The toolbars beneath the main menu are hidden as well as all main menu entries except the File menu. The graphical editor additionally shows the Properties View with only the Ontological tab since all other options are irrelevant to the user of a modeling language. This allows four different applications to be shipped for four different stakeholders based on one underlying deep model.

Chapter 11

Evaluation

This chapter evaluates to what extent the previously described technologies and their combination within one modeling approach address the seven requirements for a modeling environment presented in the introduction. It does this by using the deep modeling approach to model a case study and comparing it to other models developed using an alternative state-of-the-art modeling environment. In each of the following sections one of the seven requirements is presented in the context of the company structure modeling language used throughout this thesis. After first showing how the requirement is addressed with the technology developed in this thesis the section examines how it is addressed by the selected state-of-the-art modeling tool and then presents a comparison.

The criteria for selecting the competing state-of-the-art comparison tool for the evaluation are as follows: first, the tool has to be widely adopted and *best-in-class*, second, the complete set of requirements defined in the introduction needs to be addressable in some form, and third, the tool must be freely available to public. The last point excludes some commercial tools, but guarantees that the results of the case study can be reproduced without commercial license obstacles. Besides this in depth comparison to one tool, the related work chapter provides a more comprehensive comparison of the Melanee technology to all major academic and industrial tools available on the market at the time of writing.

The environment that is chosen is a combination of various subprojects from the Eclipse Modeling Project [66, 213] which is one of the most successful and widely distributed open source modeling platforms available at the time of writing. The different EMF subprojects used in the evaluation are presented when they first appear.

11.1 R1: Deep Modeling

The company modeling language features more than one classification level starting with a very general language specific to the domain of modeling company structures. This level of abstraction provides types for modeling different kinds of companies, departments and employees which can be instantiated to model types specific to the kind of company to be modeled. This means that company, department and employee blueprints are introduced into the language which are customized for modeling the company at hand. These blueprints are then further instantiated to model the actual company structure. The two instantiation steps described here imply three different classification levels: first, the classification level holding the very general company structure modeling language, second, the classification level holding the language which is tailored towards a specific kind of company and third, the structure of the actually modeled company.

11.1.1 Deep Model of the Company Structure Language

The realization of the deep company structure modeling language using Melanee is rather straightforward because the LML-based deep modeling approach, implementing the work of Kennel [127], natively supports modeling over an unlimited number of classification levels. In this approach each of the languages is defined at its an own classification level and model elements at one level are classified by model elements at a more abstract (i.e. *higher*) classification level. The corresponding deep model is displayed in Figure 11.1. The generic types supporting the creation of a company-specific language are placed at level O_0. These are CompanyType for the creation of different types of companies, DepartmentType for the creation of different types of departments and

several EmployeeTypes, namely ManagementEmployeeType, FullTimeEmployeeType and TemporaryEmployeeType. All of the types at this level have potency of *two* so that they can be instantiated on the following two levels resulting in a deep model with three levels in total. The only type that does not have potency *two* is EmployeeType which serves as the supertype for the different employee types. The instance of this type is an abstract supertype for the different employees modeled at level O_1. The possible combinations of the generic type's instances are indicated by the connections with potency *two*. CompanyType instances are connected with their corresponding DepartmentType instances through the departments connection and DepartmentType instances can be connected with instances of EmployeeType. Moreover, a ManagementEmployeeType is connected to all EmployeeTypes it manages via the manages connection. The generic company structure modeling types are further characterized by the definition of attributes. All attributes at level O_0 are defined with potency two so that they endure across the following two classification levels. CompanyType is characterized by a name and a legalForm (e.g. *Ltd.*). DepartmentTypes own attributes describing their name and location. Each EmployeeType is described by a name, its salary and expertise.

To create a company-specific modeling language, the generic types from the O_0 level are instantiated on the second level, O_1. A ToyCompany is defined as type representing a company creating toys. In this example a ToyCompany usually consists of two different types of Departments: one MarketingDepartment and one ResearchDepartment. Different Employees can work in these Departments. The language shown here defines Researchers, MarketingExperts, Interns and ProjectLeads managing other Employees. All clabjects and connections at level O_1 have a potency of *one* except two clabjects, which are Department and Employee with a potency of *zero*. These, therefore, play the role of abstract clabjects for their subtypes. The only clabject not typed by a clabject from O_0 is Department. Department is introduced with potency *zero* to simplify the definition of the connection between Departments and their Employees because a connection from each Department to each single Employee would have to be created otherwise. For the same reason Employee is instantiated at level O_1 as a superclass to be connected to the ProjectLead so that ProjectLeads can manage any kind of Employee. For all subclasses of Employee a default value for the expertise at-

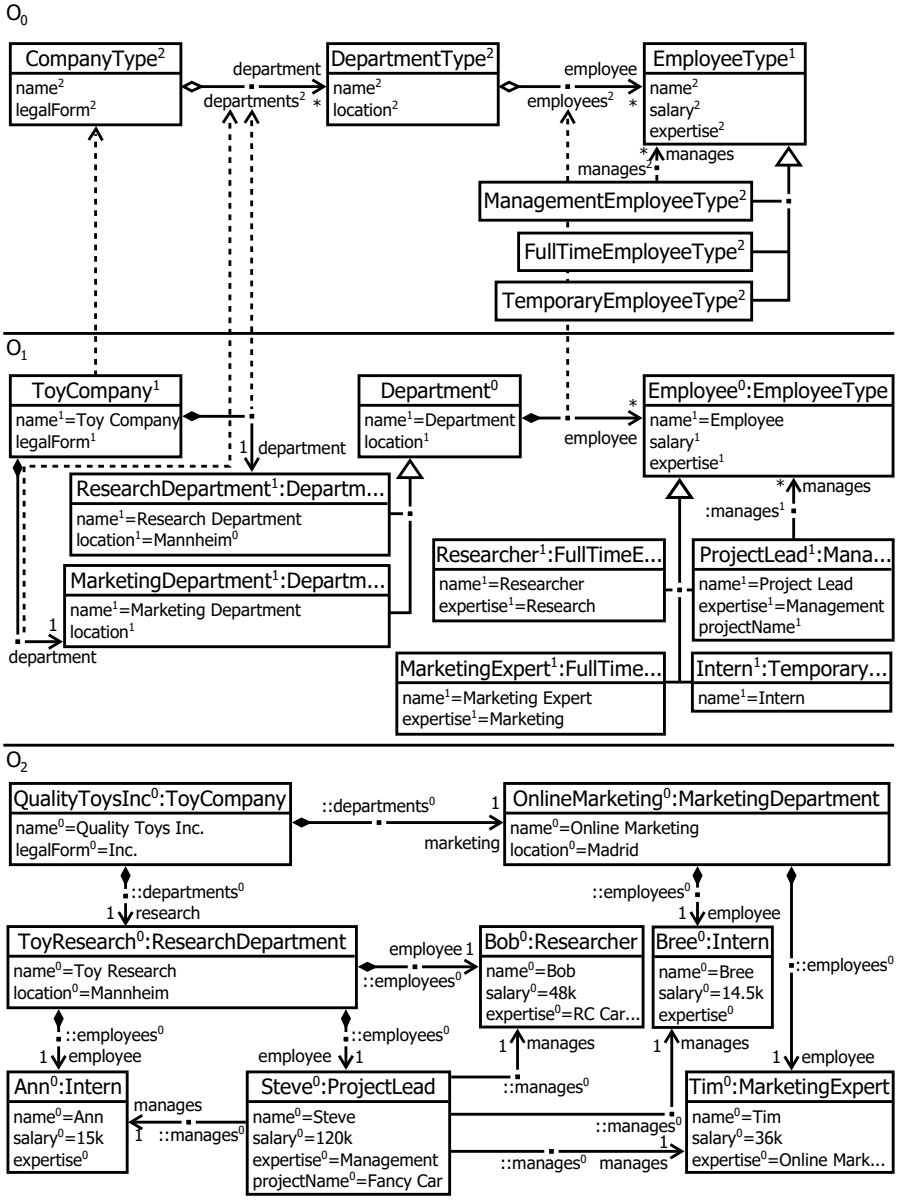


Figure 11.1: Deep Model of the Quality Toys Inc.

tribute is set. Except for Interns which do not have any expertise given that they are still learning. An attribute is added to ProjectLead naming the project which is currently being led (projectName). Also a default value for each clabject's name attribute is set to the linguistic identifier of the clabject. A special value is set for the location of ResearchDepartment, which is set to Mannheim with a mutability of *zero*. Setting the mutability to *zero* changes the nature of this value from a default value to a constant value which cannot be changed at instance levels. Hence, ResearchDepartments can only exist in Mannheim.

On O_2, the most concrete level, the QualityToysInc company is modeled using the types made available at level O_1. The QualityToysInc company consists of two departments which are OnlineMarketing and ToyResearch. The ToyResearch department is located in Mannheim which is the constant value defined by its type ResearchDepartment and the OnlineMarketing department is located in Madrid. Bob, a researcher, and Ann, an intern, work in the ToyResearch department. The expertise of Bob is RC Car Research and he has a salary of 48k. Ann does not have any expertise because she is an intern, and earns a salary of 15k. The ToyResearch department also has Steve, a project lead, responsible for managing Ann and Bob. Steve's expertise is Management, his salary is 120k and he leads the Fancy Car project. Two other employees who are also managed by Steve work in the OnlineMarketing department. These are Bree, an intern with no expertise and a salary of 14.5k, and Tim, a marketing expert with expertise in Online Marketing and a salary of 36k.

11.1.2 Non-deep EMF Model of the Company Structure Language Example

The model displayed in Figure 11.1 shows that the deep company structure modeling language can be modeled using the standard constructs of Melanee without applying any workarounds. However, modeling this scenario using a non-deep modeling language such as the EMF forces a modeler to squeeze the three classification levels of the company structure modeling language into two. Additionally, one of these two levels is *fixed* while the other is *soft* (i.e. editable) when modeling. Hence, a decision about which parts of the company structure language are hard-wired and which are soft has to be made upfront

when developing the company structure modeling language. In [151] de Lara et al. present modeling patterns which make more than one classification level available to a modeling language limited to one hard-wired and one soft-wired classification level. These patterns are applied in this context to model the deep company structure modeling language using a non-deep framework.

In the example, two patterns are identified as applicable for building a minimalistic version of the deep company structure modeling language, the *type-object* and the *dynamic features* patterns. Below these two patterns are used to build the company structure modeling language in the EMF. The choice of EMF rules out solutions involving stereotypes or powertypes since they are not fully supported in the EMF.

Type-object Pattern

To build up the structure of the company modeling language, the *type-object pattern* is employed first. All types residing at level O_0 in the deep version are put at level M_2 in Figure 11.2 and are thus hard wired into the two level company structure modeling language. These are CompanyType, DepartmentType and EmployeeType inheriting from the abstract metaclass ElementType. O_1 instances are modeled as instances of these types at level M_1. To simulate deep modeling, in this case modeling instances of instances of the types defined in M_2 (i.e. O_2 instances), the *type-object pattern* is applied. This pattern adds a metaclass representing these instances to M_2. Here Company, Department and Employee as subtypes of the abstract metaclass ElementInstance are added as types for instances of the corresponding types (e.g. instances of Company are instances of CompanyType instances). To represent this type/instance relationship in the model, the type reference pointing from ElementInstance to ElementType is introduced. This reference is redefined at each subtype to ensure that instances are instance of the correct type. Hence Company instances, for example, can only be instances of CompanyType instances. The *redefines* property is not available for references in EMF but is used here to reduce the complexity of the example. The classes representing the type instances duplicate the attributes and references defined by the classes representing their types. This allows attribute values to be set and model elements to be connected at the type and instance level in an appropriate way.

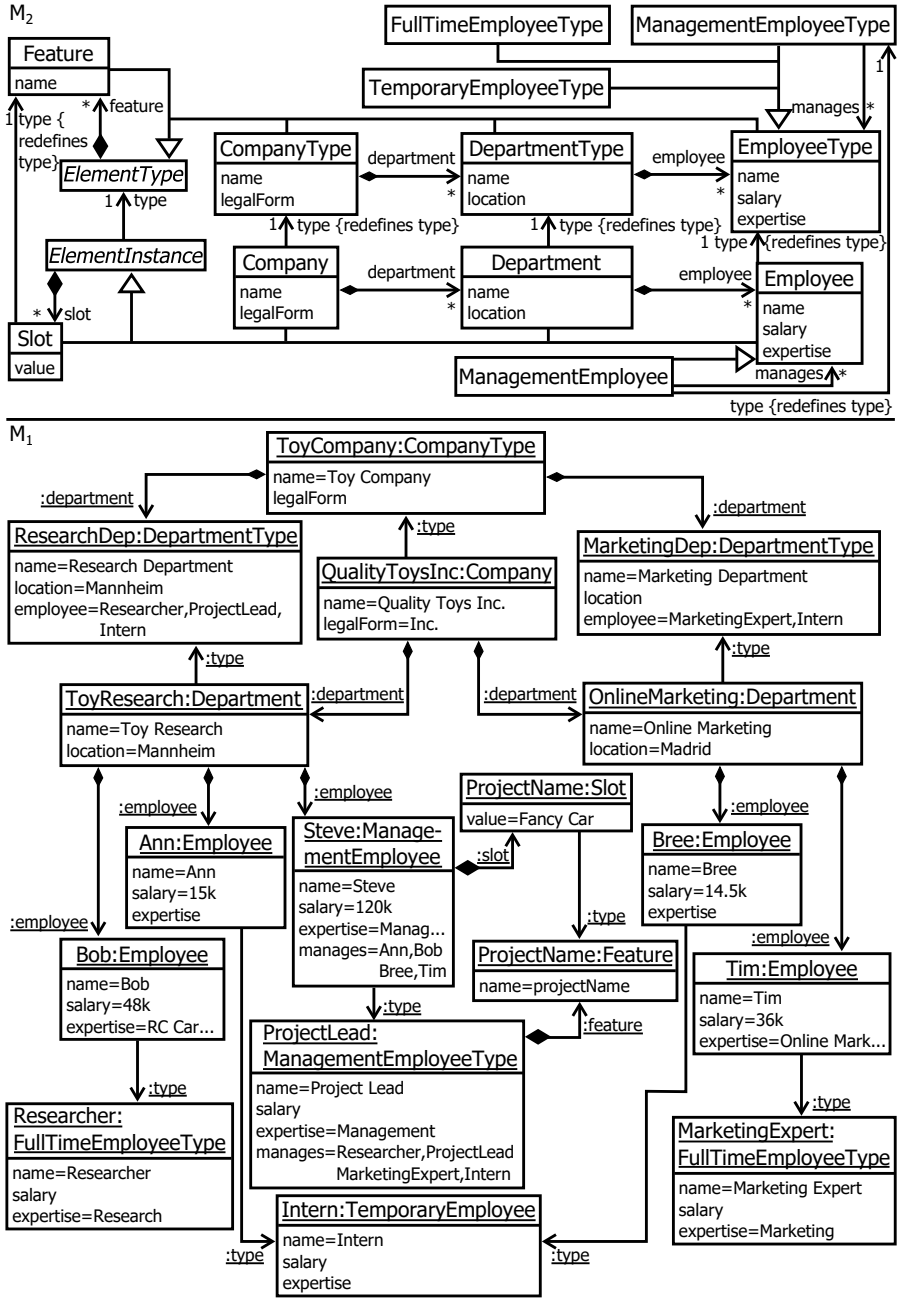


Figure 11.2: EMF Model of the Quality Toys Inc.

The different kinds of employees, FullTimeEmployeeType, TemporaryEmployeeType and ManagementEmployeeType, are modeled as subtypes of EmployeeType. In addition, ManagementEmployeeType has a manages reference to EmployeeType defined which represents all EmployeeTypes that can be managed by a certain type of manager. ManagementEmployee is added as a subtype of Employee to connect instances of ManagementEmployeesTypes to their managed Employees. ManagementEmployee redefines the type reference of Employee so that only instances of ManagementEmployeeTypes manage other employees. Additionally two OCL constraints are applied to the model. The first constraint on Employee (Constraint 11.1.1 line 1-3) ensures that no Employee instance is connected to ManagementEmployeeType via a type reference instance, i.e. only ManagementEmployees can be instance of ManagementEmployeeType. The second constraint (Constraint 11.1.1 line 4-5) ensures that ManagementEmployees manage only instances of EmployeeTypes which are also managed by their ManagementEmployeeType.

Constraint 11.1.1 (Constraints on M_2 of Figure 11.2)**.**

```
1 context Employee
2 self.oclIsTypeOf(Employee) implies
3                                   not type.oclIsKindOf(ManagementEmployeeType)

4 context ManagementEmployee
5 manages->forAll(e | type.manages->includes(e.type))
```

On the M_1 level in Figure 11.2 the O_1 and O_2 levels of the deep version are modeled. A ToyCompany is modeled together with a choice of departments (ResearchDepartment, MarketingDepartment) of which such a ToyCompany can be composed. The ResearchDepartment employs Researchers, Interns and ProjectLeads. MarketingExperts and Interns are employed in the MarketingDepartment. These O_1 types are then instantiated via the type relationships. ResearchDepartment is instantiated by ToyResearch and MarketingDepartment by OnlineMarketing. Ann, an Intern, Bob, a Researcher, and Steve, a ProjectLead work in the ToyResearch department. Whereas Bree, an Intern, and Tim, a MarketingExpert, work in the OnlineMarketing department.

The example in Figure 11.2 deviates from the deep version in that the departments and employees defined at O_1 are directly connected with each other in the non-deep version. In the deep version, however, all employees can work in all departments because Department, the supertype of ResearchDepartment and MarketingDepartment, is connected to Employee. In the non-deep version, (i.e. EMF) no inheritance is supported on the M_1 level. Hence, the departments are directly connected to the types of employees that work in them. If in the future it is intended that an instance of MarketingExpert shall work in a ResearchDepartment, these two types would have to be connected with each other on M_1. In contrast, in the deep version this is achieved by connecting supertypes.

Constraint 11.1.2 (Ensuring Correct Instance Linking)**.**

```
1 context Company
2 department->forAll(d | type.department->includes(d.type))

3 context Department
4 employee->forAll(e | type.employee->includes(e.type))
```

Based on the metamodel at M_2 of Figure 11.2, the Department and Employee instances can be connected independently of their types. For instance, Bob could work in the OnlineMarketing department even though no link between its type, Researcher, and the type of OnlineMarketing, MarketingDepartment, exists. To ensure the correct linking of instances of types defined at M_1, additional constraints shown in Constraint 11.1.2 are defined. The first constraint (line 1 - 2) ensures that instances of a Company are linked to instances of Department as defined by their types (CompanyType and DepartmentType instances) and the second constraint (line 3 - 4) ensures that instances of Departments are linked to instances of Employees to which their type (instance of DepartmentType) is linked.

Dynamic Features Pattern

In the deep version of the example a projectName attribute describing the name of the project being led is added to ProjectLead. The dynamic addition of at-

tributes to types defined in M_1 is realized through the application of the *dynamic features pattern*. By applying this pattern ElementType, representing the types at M_1, is connected to Feature representing the type facet of attributes. ElementInstance, representing the instances at level M_1, is connected to Slot representing an attribute's instance facet. Each Slot is connected to its type facet (Feature) via a type reference.

The pattern is applied to ProjectLead defining the ProjectName attribute as an instance of Feature. Hence, all instances of ProjectLead can define Slots providing a value for the managed ProjectName. Here, Steve sets the value Fancy Car for the project he manages through the ProjectName slot which is an instance of the ProjectName feature.

Constraint 11.1.3 (One Slot for each Feature)**.**

```
1 context ElementInstance
2 type.feature->forAll(f | slot.type->includes(f))
```

The metamodel at M_2 does not specify that an ElementInstance has to define one Slot for each Feature defined by its ElementType. Hence, in the example it is possible to create ProjectLead instances which do not provide a Slot for the ProjectName feature. Constraint 11.1.3 defined on ElementInstance forces an instance to have one Slot for each Feature defined at the type. The constraint first navigates to the ElementInstance's type and collects all of its defined Features (type.feature). Then, for each Feature of the type (→forAll(f|...)), it checks whether ElementInstance has a Slot with this Feature as its type (slot.type->includes(f)). This requirement is not expressible via the definition of cardinalities on references between Feature and Slot.

Application of Further Patterns

The non-deep EMF version of the deep model presented so far features everything needed to build the deep company structure example as shown in Figure 11.1. However, to simulate the full power of deep modeling, additional patterns have to be employed. These patterns, however, have not been introduced into the non-deep company structure modeling language as displayed in Figure 11.2 for space reasons. In the following sections the *dynamic auxiliary*

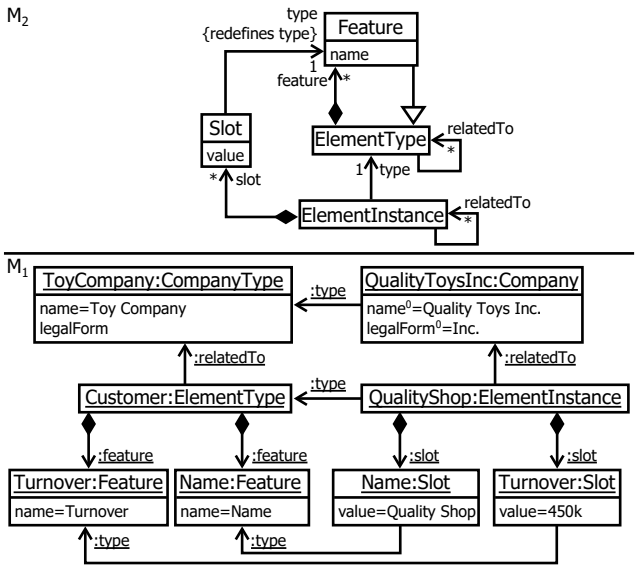


Figure 11.3: Addition of the *dynamic auxiliary domain concepts pattern*.

domain concepts pattern, for adding new types and their instances to M_1, the *relation configuration pattern*, for setting cardinalities for connections between types and instances defined at M_1 and the *element classification pattern* for introducing super and subtype relationships between types at level M_1 are introduced.

Figure 11.3 shows an excerpt of the non-deep company structure language to which the *dynamic auxiliary domain concepts pattern* is applied. To allow new types to be introduced at M_1 the metaclasses ElementType and ElementInstance are no longer abstract. Additionally, relatedTo references pointing from ElementTypes and ElementInstances to themselves are added. These generic metaclasses together with the Feature and Slot metaclasses can be used to model new types and their instances at M_1

In the example, Customers relatedTo ToyCompanies are added as instance of EntityType. The type is further described by Feature instances holding the customers name and turnover. The Customer is instantiated by QualityShop filling the Name slot with the value Quality Shop and the Turnover slot with the value 450k. The QualityShop is relatedTo the QualityToys company.

The metamodel shown at level M_2 of Figure 11.3 does not ensure that links of instances conform to the links of their type. Thus, in the example, the QualityShop could be connected to any object such as any department or employee. Constraint 11.1.4 ensures that instances of a type are linked to instances of types to which their type is linked.

Constraint 11.1.4 (Correct Linking of Dynamic Domain Concepts)**.**

```
1 context ElementInstance
2 relatedTo→forAll(r | type.relatedTo→includes(r.type))
```

Until now it is not possible to refine the cardinalities between model elements at the M_1 level which corresponds to the O_1 and O_2 levels of the deep version. Using the deep modeling approach this is possible out-of-the-box. The cardinality at any connection end can be chosen as desired and the constraint it defines is enforced across all following classification levels. To achieve the same in the non-deep version, the relation configurator pattern has to be applied as shown in Figure 11.4. In this example a class representing the connection at the type level (RelatedToType) is associated with ElementType. Using this metaclass, cardinalities (min, max) can be defined between linked ElementType instances. The connection cardinalities are enforced via Constraint 11.1.5 for all RelatedTo instances connecting instances of connected types.

Constraint 11.1.5 (Cardinality Constraints)**.**

```
1 context ElementInstance
2 type.relatedTo→forAll(type | let instanceCount:Integer =
3    irelatedTo→select(instance | instance.type = type)→size()
4    in (instanceCount ≤ type.max or type.max = -1)
5                                        and instanceCount ≥ type.min
6)
```

The constraint in the context of ElementInstance checks for all connections at the type level (type.relatedTo→ forAll()) whether the size of the instance connections (irelatedTo→select(instance | instance.type = type)→size()) is less than or equal to the max attribute value (instanceCount ≤ type.max) and greater than the min value (instanceCount ≥ type.min). Also unbounded cardinality, expressed

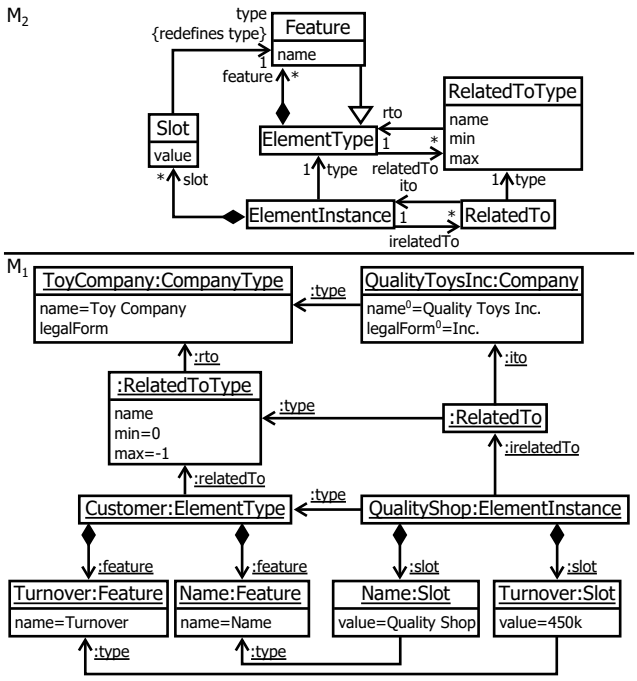


Figure 11.4: Addition of the *relation configurator pattern*.

as an upper bound of -1, is supported by the constraint (or type.max = -1). In the case of an unbound maximal cardinality the check for the upper bound is simply skipped.

The last modification to the non-deep company structure modeling language is the application of the *element classification pattern*. Until now the EMF-based version has no inheritance relationships on M_1 in contrast to the deep version. For a model of this size it is no problem to explicitly link all types at M_1 with each other. For bigger models, however, it is desirable to link supertypes and inherit links to subtypes. For this reason the *element classification pattern*, which adds references representing inheritance relationships to M_1, is applied.

Figure 11.5 shows the application of the *element classification pattern* to an excerpt of the non-deep company structure modeling language. ElementTypes can define inheritance relationships via the super/sub reference which is refined at each subtype of ElementType so that types can only inherit from the same

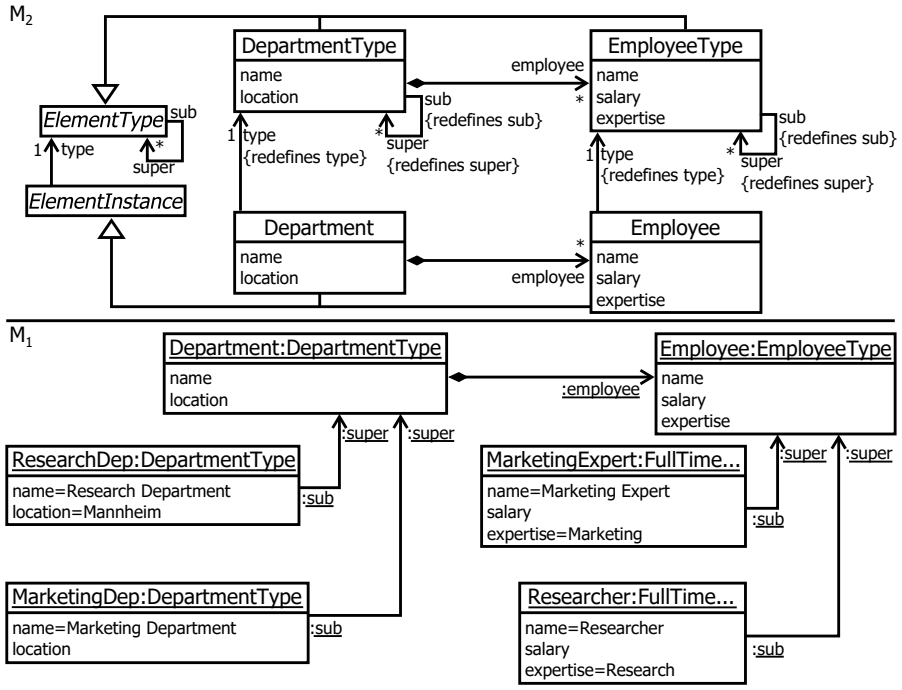


Figure 11.5: Addition of the *element classification pattern*.

type, i.e. DepartmentTypes inherit from DepartmentTypes etc. On level M_1 ResearchDepartment and MarketingDepartment use the super/sub reference to express their subtype relationship to Department and MarketingExpert and Researcher use the reference to express that they are subtypes of Employee. The two supertypes Department and Employee are connected via an instance of the employee reference, expressing the fact that all Employees can work in all Departments. Without this modification, each department type is directly connected to all types of employees working in the department. This creates complexity when adding new types of employees or when employee types can be assigned to department types to which they could not be assigned before. After applying the *classification pattern* this complexity is reduced and the deep and non-deep version of the model are equivalent in this regard.

Constraint 11.1.6 ensures that instances of types are only linked in the way defined by their type. The constraint checks for each connected employee

(employee->forAll(...)) if the Department's type or the supertypes of the type (type->append(type->closure(super)) is connected via an employee reference to the type of the connected employee or one of its subtypes (.employee→closure(sub)→ includes(e.type)).

Constraint 11.1.6 (Ensuring Correct Subtype Instance Linking)**.**

```
1 context Department
2 employee->forAll(e | type->append(type->closure(super)).employee
3                                   →closure(sub)→includes(e.type))
```

11.1.3 Metric-based Comparison for R1

To compare the deep and non-deep versions of the company structure modeling language different metrics can be applied. The first metric applied here is accidental complexity [38] followed by several metrics from the domain of object-oriented design.

Following Atkinson and Kühne [31], the accidental complexity of a model can be measured by counting the difference in the number of model elements present in alternative versions of the measured model. In this evaluation the complete deep version of the company structure modeling language as shown in Figure 11.1 and the EMF version as shown in Figure 11.2 are compared. For the comparison, the EMF version with the *type object pattern* and *dynamic features pattern* is chosen because this is the minimal EMF model which is needed to represent the same domain content as the deep version of the model. The other applied patterns are included to make the feature set of the non-deep version match that of the deep version (e.g. inheritance of clabjects at intermediate levels), but are not strictly needed to convey the same amount of information. In general, however, it can be observed that applying these patterns increases the number of model elements in the EMF version and thus the accidental complexity, impacting the result of the evaluation even more negatively for the EMF model.

The results of the accidental complexity analysis are summarized in Table 11.1 and show a clear advantage for the deep version even when compared to the minimal EMF solution. The deep version uses 23 clabjects compared to 31 classes in the EMF version which is an increase of 34.78%. The increase

Metric	Deep	EMF	Accidental Complexity	%
Class Count (DSC)	23	31	8	34.78%
Connection Count	18	47	29	161.11%
Inheritance Count	9	12	3	33.33%
Well-formdness Rules	0	5	5	—
Total	**50**	**95**	**45**	90.00%

Table 11.1: Calculation of accidental complexity to express the whole model.

in the number of classes is caused by the application of the *type object pattern* which adds one class representing the instance facet of a type to each type. Additionally, four classes are added to realize the *dynamic features pattern*. In general, however, the impact of the *type object pattern* on the number of classes becomes insignificant when the number of O_2 instances greatly exceeds the number of model elements at the upper levels, as accidental complexity is only introduced to the O_0 and O_1 levels in the example. The dynamic features pattern, in contrast, can have a more significant impact on the number of classes depending on the number of features introduced in O_1.

The increase of accidental complexity is even more dramatic when looking at the number of connections present in the deep and EMF versions of the company structure modeling language. The EMF version features 47 references compared to 18 connections in the deep version which corresponds to an increase of 161.11%. When first looking at the model, the vast number of references between model elements is not immediately obvious because some references are rendered as attributes in Figure 11.2 for reasons of readability. The increase is mainly caused by the application of the *type object pattern* which adds one reference between the type and instance facets of classes at M_2 and M_1. Additionally more relationships exist at M_1 because of the lack of inheritance at this level. The introduction of the *classification pattern* as previously shown would not dramatically improve this number because it introduces additional references and constraints on M_2 for defining the inheritance capabilities of the company structure modeling language and additional links on M_1 for modeling the inheritance relationships through references. The pattern, however, can decrease complexity in scenarios with a large number of

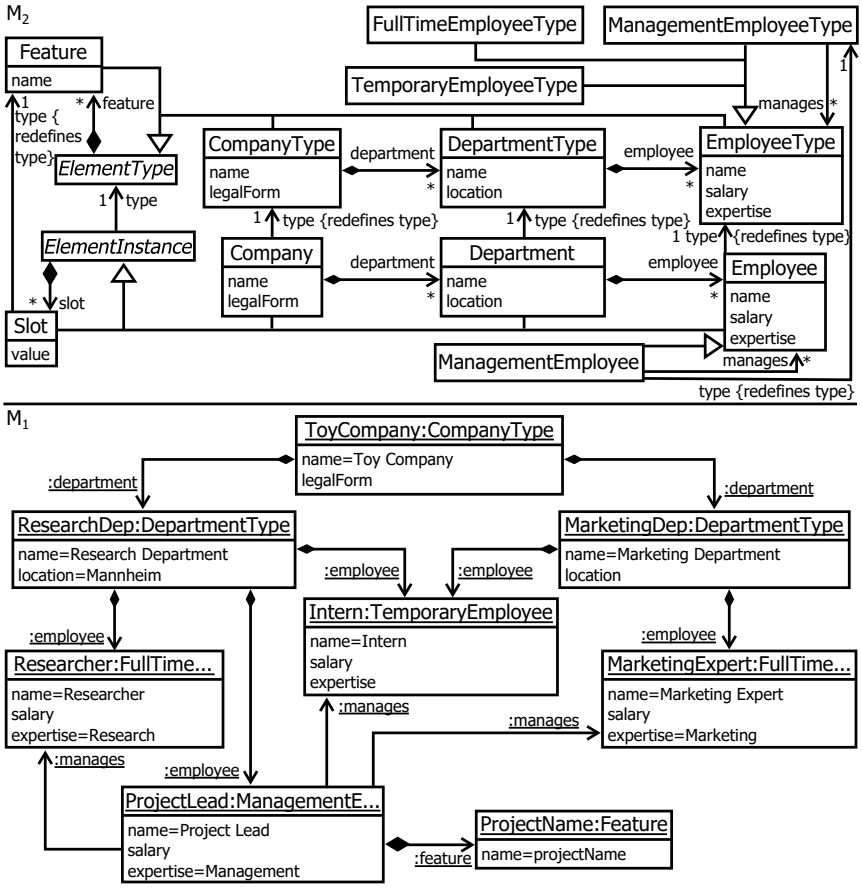


Figure 11.6: EMF model corresponding to O_0 and O_1 of the deep company structure modeling language.

inter-connected model elements at O_1.

Also, the number of inheritance connections defined in the EMF version is 33.33% higher (9 deep model, 12 EMF) and the number of well-formedness constraints in the EMF version is five compared to zero in the deep version. In total the deep version is modeled using 50 modeling constructs, whereas the EMF version is modeled using 95 modeling constructs. This corresponds to a difference of 45 modeling constructs and an additional overall accidental complexity of 90%.

In terms of accidental complexity, therefore, the deep version has a clear

Metric	Description
DSC	number of clabjects/classes
NOH	number of inheritance root classes
ANA	average number of subclasses
ANDC	average number of distinct connected classes (outgoing references)
ANAG	average number of compositions
ADI	average inheritance tree depth
NAC	number of abstract (potency zero) clabjects
ANAT	average number of attributes
AWF	average number of well-formedness rules
AAP	average number of additional operations

Table 11.2: Object-oriented metric descriptions.

advantage over the EMF version. Besides the accidental complexity metric it is also possible to apply object-oriented design metrics [46, 158, 198] and quality attributes [33] to metamodels as shown by [160]. In this evaluation, the quality attributes are calculated for the O_0 and O_1 levels of the deep company structure modeling language (Figure 11.1), because these two levels define the company structure modeling language used for modeling a specific company, i.e. Quality Toys Inc. in the example. The corresponding classes of the EMF version are shown in Figure 11.6.

The metrics measured on the deep and EMF version of the metamodel are described in Table 11.2. The descriptions show that all metrics are determined by counting certain types of modeling constructs. The DSC metric, for example, counts all clabjects/classes, and the NOH metric counts all classes/clabjects which are the root of an inheritance hierarchy. The metrics are assigned to design properties in Table 11.3 according to a one-to-one mapping except in the case of the Complexity design property which is a composition of the average of the well-formedness rules defined per class/clabject and the average of the additional operations per class/clabject. For two design properties from the object-oriented design space, Encapsulation and Cohesion, no metric is calculated following the approach of [160]. These two concepts have been assigned a default value of *one*.

Design Property	Metric	Deep	EMF
Design Size (DeS)	DSC	15	22
Hierarchies (Hier)	NOH	3	2
Abstraction (Abs)	ANA	3	3
Encapsulation (Enc)	—	1	1
Coupling (Coup)	ANDC	0.33	1.09
Cohesion (Coh)	—	1	1
Composition (Compo)	ANAG	0.33	0.64
Inheritance (Inh)	ADI	1	2
Polymorphism (Pol)	NAC	2	2
Messaging (Mes)	ANAT	1.73	1.59
Complexity (Compl)	$\frac{AWF+AAP}{DSC}$	$\frac{0+0}{15} = 0$	$\frac{5+0}{22} = 0.23$

Table 11.3: Design properties of the modeling language definition (O_0 + O_1).

Comparing the metrics and, thus, the design properties in Table 11.3 shows that both versions of the company structure modeling language are quite similar. The most significant differences are the design size which is 15 for the deep model compared to 22 for the EMF model, a coupling of 0.33 for the deep model compared to 1.09 for the EMF version, a *composition* of 0.33 for the deep model compared to 0.64 for the EMF version and a complexity of 0 for the deep model compared to 0.23 for the EMF version. The difference in design sizes can be explained by the increased accidental complexity produced by applying the *type object pattern* (addition of ElementType, ElementInstance, Company, Department, Employee) and the *dynamic features pattern* (addition of Feature, Slot) to the EMF model to allow the deep company structure to be modeled. The increase in coupling is caused amongst other things by the *type object pattern* which assigns each instance metaclass a second reference to its type metaclass at M_2, e.g. the reference between Company and CompanyType. In general, references have to be used to express language constructs which in deep modeling are present out-of-the-box, raising the number of defined references per class. The complexity difference arises from the fact that the *type object pattern* and *dynamic features patterns* are accompanied by constraints ensuring well-formedness rules which cannot be expressed through metamod-

Quality Attribute	Deep	EMF
Reusability $(-\frac{1}{4}Coup + \frac{1}{4}Coh + \frac{1}{2}Mes + \frac{1}{2}DeS)$	8.53	11.77
Flexibility $(\frac{1}{4}Enc - \frac{1}{4}Coup + \frac{1}{2}Compo + \frac{1}{2}Pol)$	1.33	1.30
Understandability $(-\frac{1}{3}Abs + \frac{1}{3}Enc - \frac{1}{3}Coup + \frac{1}{3}Coh - \frac{1}{3}Pol - \frac{1}{3}Compl - \frac{1}{3}DeS)$	-6.16	-8.82
Functionality $(0.12 * Coh + 0.22 * Pol + 0.22 * Mes + 0.22 * DeS + 0.22 * Hier)$	4.90	6.19
Extendability $(\frac{1}{2}Abs - \frac{1}{2}Coup + \frac{1}{2}Inh + \frac{1}{2}Pol)$	2.84	2.96
Effectiveness $(\frac{1}{5}Abs + \frac{1}{5}Enc + \frac{1}{5}Compo + \frac{1}{5}Inh + \frac{1}{5}Pol)$	1.47	1.73

Table 11.4: Quality attributes of the modeling language definition (O_0 + O_1).

eling concepts. Other patterns, not used in the model displayed in Figure 11.2, such as the *relation configurator pattern* and *element classification pattern* increase, amongst others, the number of constraints, and thus the complexity design property, even more when applied. Hence, a correlation between deepness of the EMF model and the resulting complexity can be observed.

The measured design properties are used to create indicators which measure about the quality of the deep and EMF version of the company structure modeling language. The criteria are: 1. Reusability — the degree to which a model can be reapplied to a new problem without significant effort, 2. Flexibility — the ability to which a model can be adapted to provide functionality for related capabilities, 3. Understandability — the degree to which the metamodel can be easily learned and comprehended, 4. Functionality — the degree of responsibility assigned to one metaclass, 5. Extendability — the availability of concepts for introducing new requirements into the design and 6. Effectiveness — the ability of a metamodel to achieve the desired functionality and behavior using metamodeling concepts. [33, 160]

Table 11.4 compares the quality attributes of the deep and EMF versions. Since both versions are constructed to have roughly the same feature set the

metrics for the languages are similar. The most significant deviation between the deep and EMF version is in the Reusability, Understandability, and Functionality quality attributes.

The Reusability metric as calculated in Table 11.4 assumes that a metamodel consisting of more classes is more reusable than one consisting of fewer classes. This may hold true when comparing two models created using the same modeling paradigm. In this case, however, the higher number of classes in the EMF version is caused by accidental complexity produced by having to simulate a deep version of a model with more than two classification levels using a technology limited to two classification levels only. When removing the accidental complexity from the calculation and setting the design size of the EMF version to the design size of the deep version a Reusability value of 8.27 is calculated which is slightly lower than the value of the deep version (8.53). This value is more realistic since the feature sets of both models support the same level of re-applicability to new situations. Whether the nature of the modeling language (i.e. being deep) has a positive effect on Reusability and should therefore be considered in the calculation of this quality attribute is a question for future research.

The difference in Understandability is caused by the application of workarounds for emulating deep modeling in the EMF version. The metric is mainly influenced by the increase in accidental complexity which is created through the introduction of additional metaclasses that negatively influence the design size and the introduction of new references on the type and instance levels to mimic deep typing, that negatively influences coupling. All other ingredients of Understandability do not significantly differ. Hence, the decreased Understandability of the EMF version goes hand-in-hand with its increased accidental complexity.

The final deviating quality attribute is the Functionality which again is positively impacted by the accidental complexity created when emulating deep modeling since the design size positively impacts this indicator. When calculating the Functionality value using the design size of the deep version, and thus ignoring the accidental complexity of the EMF version, the Functionality indicator is 4.65 which is close to the value of the deep version of 4.90.

11.1.4 Summary R1: Deep Modeling

To evaluate the advantages of deep user-defined languages over non-deep user-defined languages, the company structure modeling language was modeled completely in the deep LML-based Melanee approach and EMF. The deep model version leveraged all the deep modeling features to model the company structure modeling language without application of any workarounds. All model elements are placed at their natural classification levels and all features such as instance checking, constraint language etc. are available on the deep company structure modeling language out-of-the-box. In contrast, the lack of more than one type/object level pair in the EMF version of the company structure modeling language forces the application of workarounds as defined by de Lara et al. in [151] in order to emulate deep modeling features. By applying these patterns the deep company structure modeling language can be modeled with the non-deep EMF framework at the cost of the application of workarounds to simulate deepness in the underlying metamodeling framework (i.e. EMF) and the *deepness* simulation of out-of-the-box language features such as type checking and constraint languages.

To objectively judge the two models regarding their quality, they were compared using different metrics. The accidental complexity metric measures which version introduces the most complexity due to the inclusion of additional model elements, while the object-oriented design metrics (reusability, flexibility etc.) ensure that both models have the same level of expressiveness and are thus comparable.

The accidental complexity metric shows a clear advantage of the deep version of the company structure modeling language over the non-deep version. In total, the deep version has 90% fewer model elements than the non-deep version which is clearly significant. In theory, the worst case performance that a deep modeling language can have compared to a non-deep modeling language is an accidental complexity of 0 since deep modeling languages have all the features of non-deep modeling languages but have extensions which provide clear advantages in deep modeling scenarios.

The object-oriented design quality attributes were equal in all but three of the cases. Two of these are explained by the positive impact of model size on the metrics. The model size, however, is caused by accidental complexity

and does not show any advantage for any of these quality attributes. Thus, the metric calculations ignored the introduced accidental complexity and set the design size of the non-deep versions to the design size of the deep version. In these calculations the quality attribute indicators are close to equal. This shows that the two models used for the accidental complexity evaluation were indeed comparable. The difference in the understandability quality attribute is explained by the increased accidental complexity.

Bansiya et. al. [33] state that the weights and impacting factors of the quality attributes can be modified to fit the exact needs of an evaluation. This is not done here to avoid choosing weights in a way that gives an advantage to one of the solutions, and thus maximize objectivity. For future research the use of weights to accurately reflect the advantages of a modeling approach could be evaluated, helping to compare solutions in different modeling approaches.

11.2 R2: Seamless Modeling

The seamless modeling feature is evaluated by defining a set of changes covering the previously presented modeling operations: 1. creating an entity, 2. deleting an entity, 3. changing an entity and 4. moving an entity. The changes executed on the models during evaluation are listed in Table 11.5. These changes reflect standard modeling operations executed several times on a model during its life cycle. Examples of such operations are a change to the potency of a clabject (C5) or the addition of attributes (A3). For a better overview, the changes are grouped into the model manipulation categories and have an ID assigned.

The first group of changes tests the impact of adding new types at different classification levels (A1 - A3) and addition of new attributes at different classification levels (A4 - A5). First, a new type of employee, ExternalEmployeeType, is added to O_0 which represents employees working in a company who are employed by a third party (A1). Then, a QualityExpert which is a FullTimeEmployeeType instance and subtype of Employee is added to O_1 (A2). Cora at O_2 instantiates this QualityExpert (A3). After adding the new types, new attributes are assigned. First, a gender attribute is added to EmployeeType (A4). Then, an errorRate is added to QualityExperts describing the rate of errors made by the

Category	ID	Modification Operation
Add		
	A1	ExternalEmployeeType at O_0
	A2	QualityExpert as instance of FulltimeEmployeeType at O_1
	A3	Cora as instance of QualityExpert at O_2
	A4	Gender attribute for EmployeeType at O_0
	A5	ErrorRate attribute for QualityExpert at O_1
Remove		
	R1	TemporaryEmployeeType at O_0
	R2	Researcher at O_1
	R3	Expertise attribute of EmployeeType at O_0
	R4	ProjectName attribute of ProjectLead at O_1
Change		
	C1	EmployeeType to EmployeeBaseType at O_0
	C2	ProjectLead to ProjectManager at O_1
	C3	Potency of ManagementEmployeeType to *3*
	C4	Salary attribute to yearlySalary at O_0
	C5	ProjectName attribute to projectTitle at O_1
Move		
	M1	Expertise attribute to FullTimeEmployeeType at O_0

Table 11.5: Change set applied for seamless modeling evaluation.

QualityExpert.

The changes concerned with the remove operations first remove the TemporaryEmployeeType clabject from O_0 (R1) and then the Researcher clabject from O_1 (R2). Afterwards the expertise attribute is removed from EmployeeType (R3) and the projectName attribute is removed from ProjectLead (R4).

Changes are first made to EmployeeType which is renamed to EmployeeBaseType (C1) and then ProjectLead is renamed to ProjectManager. The potency of ManagementEmployeeType is then set to three to enable modeling at a further ontological classification level (C3). In a real world scenario, this one change would be part of a series of changes increasing the potency of all types present at O_0. Finally, the salary attribute is renamed to yearlySalary (C4) and the pro-

jectName attribute to projectTitle (C5). The modification set includes only one move operation which is the movement of the expertise attribute from EmployeeType to FullTimeEmployeeType (M1).

11.2.1 Applying Changes to the Deep Model

The application of the change set to the deep version of the company modeling language is shown in Figure 11.7. The changes are marked as gray circles with the change ID in their center at the point where the change is made. The change implications are not indicated to avoid cluttering the diagram. It can be observed that all changes take place at the ontological classification levels (O_0 - O_2). Changes at the ontological classification levels take immediate effect in a deep model. Some changes do not need a modeler to take action in order to keep classification semantics intact while others do. The emendation service supports the user in cases where action to fix the classification semantics has to be taken.

Changes without an effect on the classification semantics are the additions of ExternalEmployeeType (A1), QualityExpert (A2) and Cora (A3). The newly added types are immediately available for modeling on all other classification levels. The removal of TemporaryEmployeeType (R1) and Researcher (R2) do not force the modeler to take action either. The types are removed and the classification relationships for all their instances are lost. However, if desired, the instances can be retyped by introducing new classification relationships. This task could be taken over by the emendation service on big models to save modeling effort. The renaming of the clabjects EmployeeType (R1) and ProjectLead (R2) does not have any impact on classification relationships and thus no actions have to be taken. The changed names are immediately available for use on all classification levels.

All other changes from the change set as shown in Table 11.5 do influence classification semantics and are thus accompanied by actions taken through the emendation service. The addition of the gender attribute to EmployeeType (A4) and of the errorRate attribute to QualityExpert involve the addition of attributes to the instances of the types to which the attributes are added to fix violations of the *Attribute Correctness* well-formedness constraint. In case of gender this would include the instances of EmployeeType which are Employee, Ann, Steve,

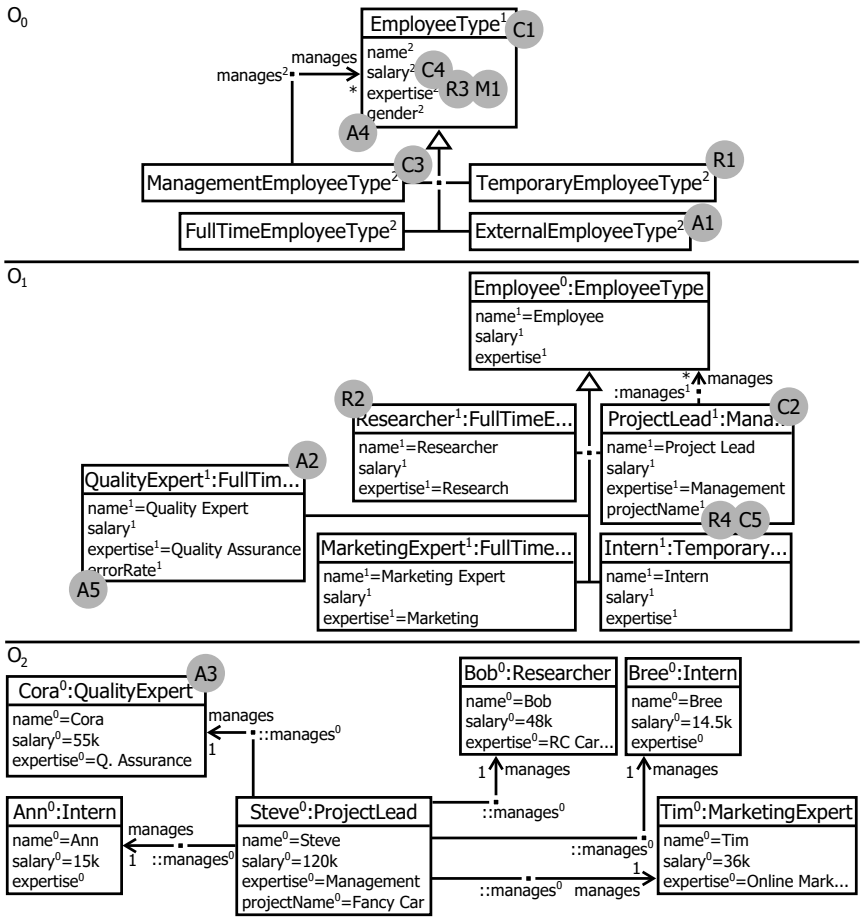


Figure 11.7: Changes to the deep model of the Quality Toys Inc.

Bob, Bree, Tim and Cora if she was already instantiated. Adding the attribute to the subclasses of Employee is optional as the attribute is inherited from Employee. When adding the errorRate attribute to QualityExpert action is only taken if Cora exists at the time of the attribute creation. If Cora exists the attribute is added.

When removing the expertise attribute from EmployeeType (R3) and the projectName attribute from ProjectLead (R4) the attribute can optionally be removed from the instances of these types as well if instances shall keep their isonym nature after the change. In the case of expertise the attribute can be

removed from Employee, Researcher, ProjectLead, QualityExpert, MarketingExpert, ProjectLead,Intern, Ann, Steve, Cora, Bob, Bree and Tim. The large number of clabjects to be modified shows that this task is hardly manageable without using the emendation service. When removing projectName only Steve is modified if desired. If the projectName existed in the type hierarchy of ProjectLead it would have to be removed from the types in order to maintain *attribute correctness*. Otherwise ProjectLead would have less attributes than its types.

The change operations effecting classification semantics are the change of the potency of ManagementEmployeeType to 3 (C3). This change increases the potency of ProjectLead and Steve which is handled by the emendation service. The renaming of the salary and projectName attributes also effects instances. When changing the name of the salary attribute all salary attributes of instances of EmployeeType are changed which are previously listed for change R3 to maintain *attribute correctness*. The change of projectName to projectTitle only effects Steve. Both change operations are again supported by the emendation service.

The move of the expertise attribute from EmployeeType to FullTimeEmployeeType is a composition of an addition operation and a removal operation which is also supported by the emendation service. After the change, the expertise attribute is removed from all EmployeeType instances except from those that are instances of FullTimeEmployeeType. Additionally the attribute is added to FullTimeEmployeeType instances where needed. In the example, all FullTimeEmployeeType instances inherit the expertise attribute from Employee and thus need to have the attribute added, if not already present, once it is removed from Employee which is not an instance of FullTimeEmployeeType.

11.2.2 Changes to the Non-Deep EMF Version

The indication of changes from the change set presented in Table 11.5 on the non-deep version of the company structure modeling language in Figure 11.8 shows that two meta levels are involved in the changes. One is meta level M_2 which is hard coded in the modeling tool and is effected by A1, A4, C1, C3, C4, R1, R3. The other is M_1, which is soft (i.e. data from the tool's point of view), and is effected by A2, A3, C2, C5, R2 and R4.

To make changes to the M_2 level in EMF it is necessary to switch from

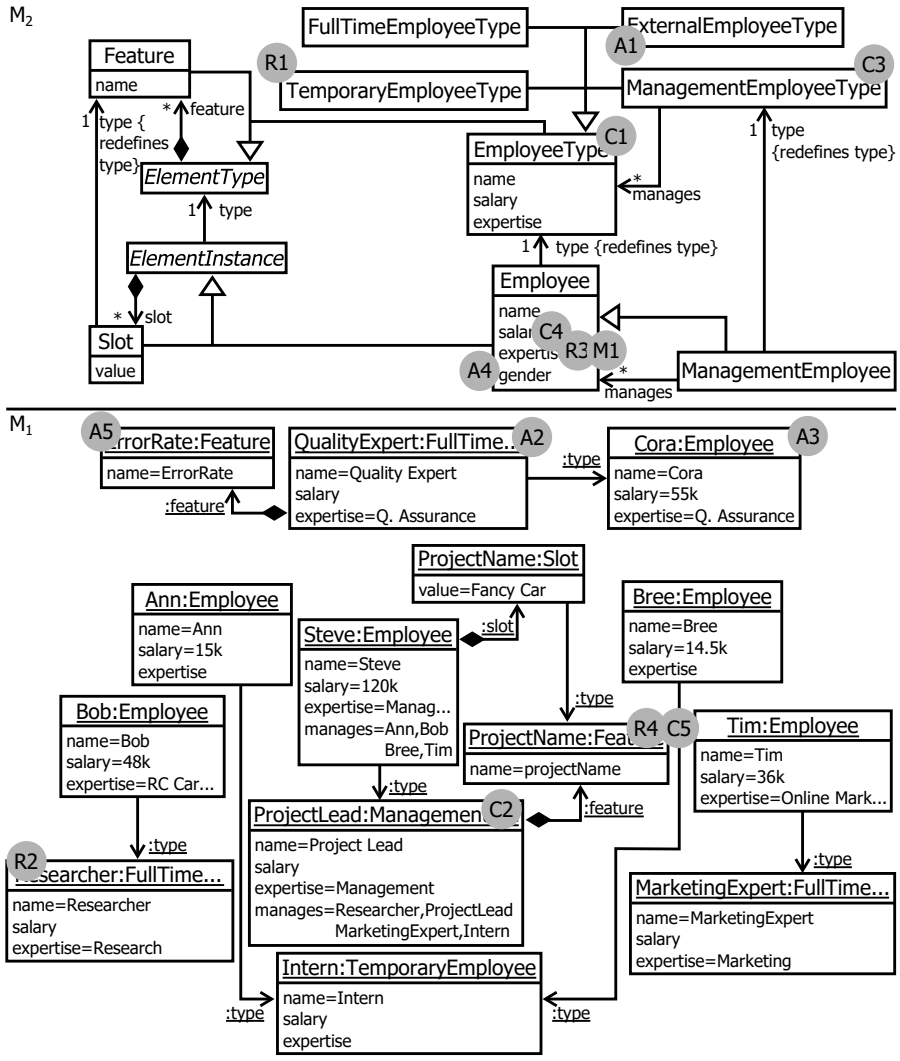


Figure 11.8: Changes to the non-deep model of the Quality Toys Inc.

the modeling environment to the tool development environment. In the development environment the metamodel is adapted and then redeployed as an update for all modeling environments. The modeling environment then applies the new metamodel. Depending on the change the already created models may no longer conform to the metamodel and are thus not usable anymore. These change are C1, C4, R1, R3, M1 in the case of the change set applied in this evaluation. The only changes which do not have this effect on the M_2 instances are A1, A4 and C3 (potency change) as these only add new modeling constructs. To handle metamodel changes different tools are available for the EMF. These include, amongst other things, a fuzzy parsing approach to model loading, [141] which can deal with a huge number of changes to metamodels or an operation recording approach [108, 68] which applies recorded operations to model instances. The operation recording approach works in a similar way to the emendation approach presented for deep model evolution. The difference is that the operations are collected and applied to the decoupled model instance. The applied operations themselves, however, are identical to the emendation service operations.

The only change at M_2 which is executed differently from the deep version of the company structure modeling language is the increase to the potency of ManagementEmployeeType. To emulate this potency increase an additional metaclass associated with ManagementEmployee through a type reference has to be added to M_2. In contrast to the deep version where the potency change has to be backed up by the emendation service to keep classification semantics intact, no further actions are needed in the EMF version.

Changes at the M_1 meta level are not supported by any out-of-the-box EMF project. Changes maintaining classification semantics thus have to be executed by hand or M_1-specific tooling that supports such changes has to be developed. Changes that need additional actions are A5, R4 and C5. Changes which do not need any attention are A2, A3, C2 and R2 in case the modeling language would regard an untyped Bob as correct.

To handle A5, which is the addition of the errorRate attribute to QualityExpert, a slot typed by errorRate has to be added to Cora so that the QualityExpert instances have one Slot for each Feature of their type. The removal of the projectName feature (R4) requires all Slots which are typed by it to be removed

so that no instance has a slot which is untyped. Depending on constraints defined on the Feature and Slot metaclasses it might be necessary to rename all Slots typed by projectName after executing the rename operation (C5).

11.2.3 Summary R2: Seamless Modeling

The evaluation shows that in deep modeling all changes defined in the change set are all equally supported across all ontological classification levels. In particular, a modeler does not need to change the modeling environment to edit certain parts of a model and all changes are immediately available for further modeling without any deployment steps. The emendation service is able to back up all changes defined in the change set because the deep classification semantics is defined on the linguistic metamodel which spans all ontological classification levels. On the other hand, the evaluation shows that even small changes have many knock-on changes which are hardly manageable without an emendation service.

In the EMF version, in contrast, one subset of the changes takes place at the M_2 level and the other subset at the M_1 level. The classification semantics between M_2 and M_1 is clearly defined and thus supported by EMF tools out-of-the-box. Hence, deep modeling and EMF are aligned when changes are made between M_2 and M_1 (corresponding to changes at O_0 of the deep version). However, changes to the M_1 level (corresponding to O_1 and O_2 of the deep version) which effect the artifacts simulating deepness are not supported by any EMF tooling out-of-the-box. To support such changes custom model evolution support would have to be developed based on the applied deep modeling patterns. This is a clear disadvantage compared to deep modeling which can support modifications across all classification levels in an equal way through the emendation service.

Changes to the meta levels M_2 and M_1 are executed in two different modeling environments. To edit the M_2 level, the modeling environment has to be switched to the metamodeling environment and the changes made to the M_2 level have to be propagated to the modeling tool (M_1) after modification of M_2. This is not the case for deep modeling where all levels are equally available and no deployment steps are needed between ontological classification levels after model modification. Changes at the M_1 level are immediately available for

modeling in EMF as is the case for deep modeling. Another disadvantage of EMF is that M_1 models which do not conform to M_2 anymore are not useable until either a manual or automatic model evolution operation is applied to the M_1 level. The deep model still works with non-conforming levels, but would show validation errors.

A problem which is not directly addressed in this evaluation is the impact of model changes on concrete syntax and their accompanying constraint definitions. In Melanee, concrete syntax definitions and their supporting constraints are directly defined within the deep model. In the current version, the concrete syntax definitions use pointers to the deep model wherever deep model content is referenced, but the constraints supporting the concrete syntax definition do not. In future implementations it is planned to use pointers when referring to model elements in a deep model from these constraints. These are then automatically updated when changing the deep model and the emendation service can even be extended to fully support the handling of the impact of abstract syntax changes to concrete syntax and constraints. To realize this, constraints have to be stored in a half-parsed format in which all text pointing to deep model content is replaced by a pointer to the corresponding deep model content.

The EMF version, however, does not support co-evolution of abstract and concrete syntax in the way deep modeling does. Concrete syntax is defined in either text (e.g. XText, Parsley) or models (e.g. GMF, Sirius) which are both stored independently of the abstract syntax definition. Hence, these models do not evolve together with the abstract syntax and additional steps as e.g. described in [58] have to be performed.

11.3 R3: Multi-notation Modeling

To evaluate the multi-notation requirement, three diagrammatic notations are created. Even though the evaluation is limited to diagrammatic notations the results can be transferred to other formats because the strengths and weaknesses relate to the general concepts on which the modeling workbenches are based rather than the specific details of particular formats.

The first notation chosen for the evaluation is a UML like *instance specifi-*

cation notation which views model content in a class diagram like view. The second is the full version of the previously introduced company structure modeling language (*private notation*) and the third is a version of the company structure language that does not contain sensitive employee-related information (*public notation*). Hence, the salary and expertise of employees is not shown in this notation.

11.3.1 Deep Notation Definition

In the deep modeling approach a notation is defined by attaching several visualizers to the clabject types. When visualizing a model the visualizer search algorithm searches a visualizer for each clabject. If no visualizer is found or the user decides not to use a visualizer, the predefined LML notation is used to represent the clabject. This already fulfills the requirement for the first notation identified in the evaluation, a UML class diagram like *instance specification notation*.

The other two diagrammatic notations, public and private, are defined using visualizers as previously described. The private notation shows all employee information whereas the public notation hides sensitive employee information. Figure 11.9 shows the definition of the two user-defined visualizations. The visualizers defined for CompanyType and DepartmentType contribute to the private and public notation because there is no difference between the displayed information in the two notations. EmployeeType has two visualizers attached, one for the private and one for the public notation.

On the O_2 level the different notations are used to visualize the company structure of the Quality Toys Inc. company. The figure shows that the different notations can be mixed and matched as needed. Steve is visualized using the predefined out-of-the-box, UML class diagram like LML notation, Bree is visualized in the public notation whereas Ann, Bob, and Tim are visualized in the private notation.

Even though a mix of different notations in one level is shown in Figure 11.9 it is possible to switch the whole level into one single notation by a simple notation toggle operation. This is a more likely scenario in this example because a user typically wants to see all information about all employees or to hide all sensitive information for all employees. In addition it would be possible to use

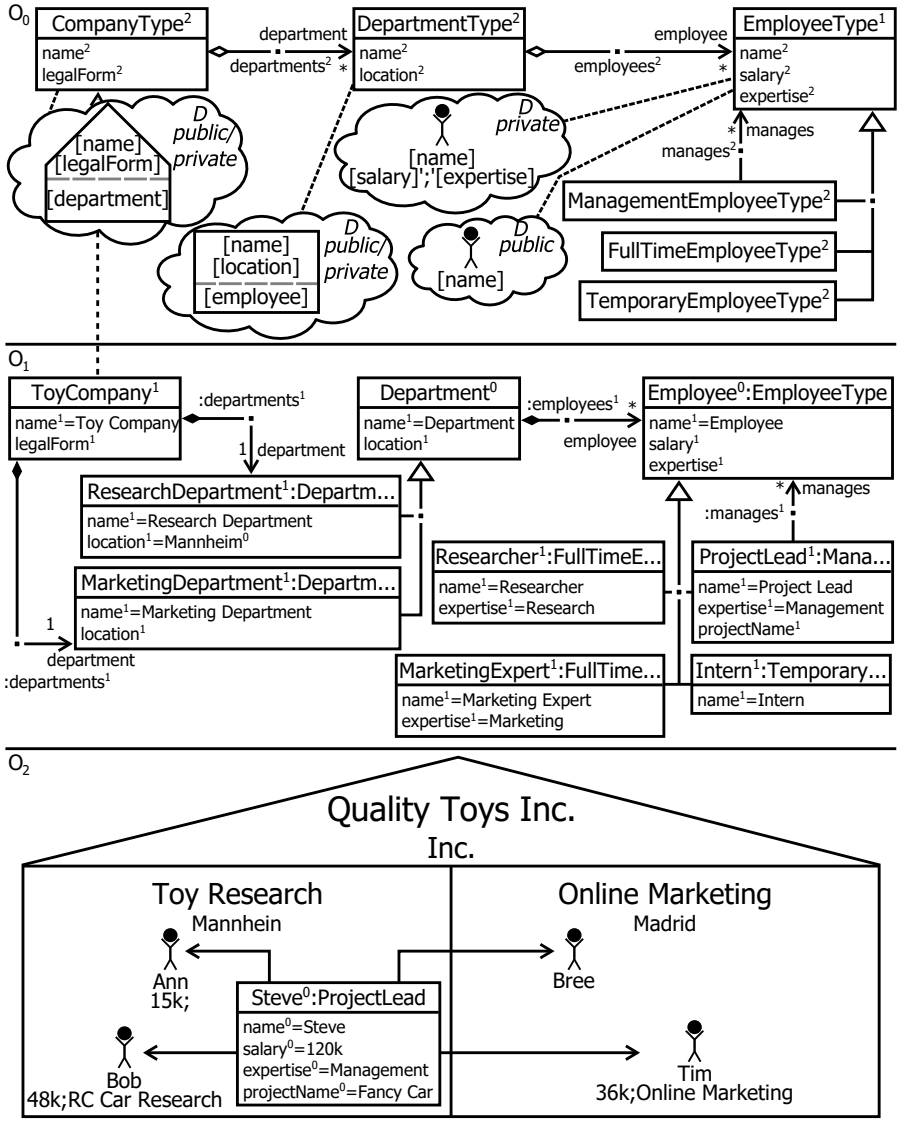


Figure 11.9: Notation definition for the deep model of the Quality Toys Inc.

the languages defined at O_0 to model at O_1. Moreover, the notation can be refined across the other classification levels.

11.3.2 Non-deep Notation Definition

To define the three notations in EMF a concrete syntax modeling framework has to be chosen. For the evaluation, GMF was chosen because it is model-driven and offers high flexibility through its customizable code generation capabilities. EMF and GMF do not ship with any predefined diagrammatic language for visualizing model instances, so three concrete syntaxes are defined here. These are the *instance specification notation*, the *private notation* including all employee details and the *public notation* excluding employee details.

To define a notation in GMF four models are created, a *tooling model*, a *graphical definition model*, a *mapping model* and a *generator model* in addition to EMF's *abstract syntax model*. From these models source code containing a notation-specific instance editor is generated. The *tooling model* defines the palette available for modeling and is thus an ideal model to be shared between all three notations since they all visualize the same language. The *graphical definition model* is similar for the *private* and *public* notation except for the employee figure which has to be defined once for the public and once for the private notation. Hence, this model can also be shared between the two notations. A new model is created for the UML instance specification like notation. The concrete syntax defined in the three *graphical definition models* is then mapped to the *abstract syntax model* and the *tooling model* by three different *mapping models*, one for each notation. These mapping models are then used to create three different *generator models* which generate the source code of the notation specific editors.

Figure 11.10 gives an overview of all models participating in the definition of the three notations. Models are solid boxes with their name located at the bottom except for the tooling model displaying its name at the top. Links between models are represented by dashed/dotted lines and in case of the tooling model by a dashed background mapping one tool to the content of three mapping models which in the case of the example contain one mapping to Employee only. The Graphical model in the upper left defines the graphical

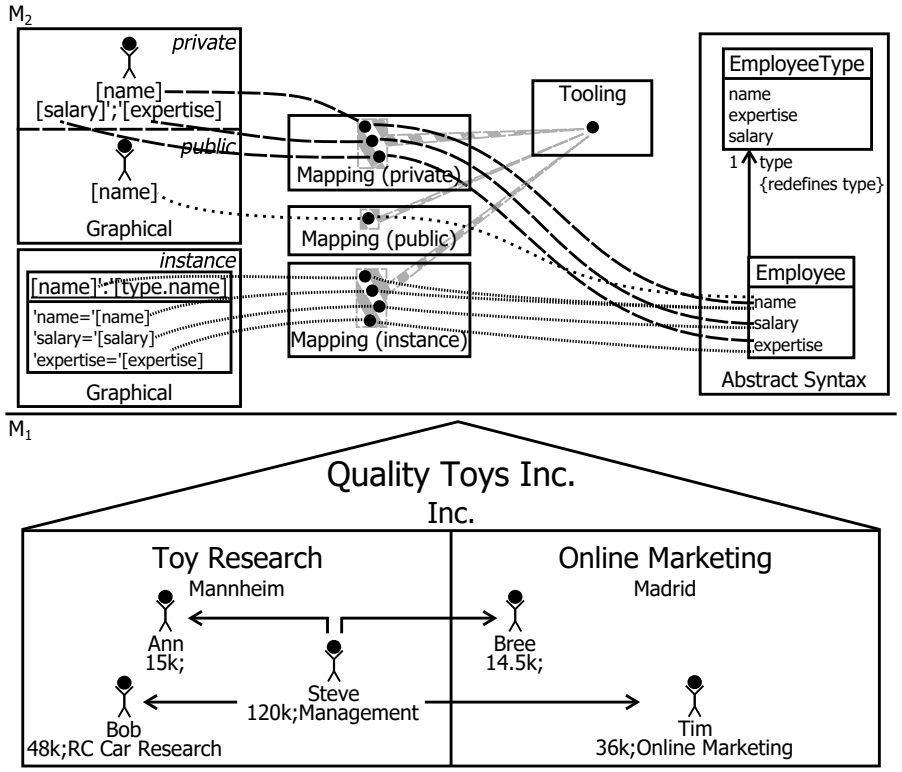


Figure 11.10: Notation definition for the non-deep model of the Quality Toys Inc.

shapes for the user-defined notation of the deep company structure modeling language. One shape is defined showing the salary and expertise of an employee and one hiding this information. These two graphical shapes are then mapped onto Employee via two different mapping models — the private notation and public notation mapping models. The tooling model is for both notations the same because the concepts to be added to a model are equal in both cases.

The shapes for the UML instance specification like language are modeled in the graphical definition model designated as instance in Figure 11.10. The instance mapping model maps the shapes to same tooling model which is used for the public and private notations and on Employee defined in the abstract syntax.

For each mapping model one *generator model* is generated which is not

displayed in Figure 11.10 for space reasons. The *generator models* combine all information from the *graphical definition models*, *mapping models*, *tooling model* and *abstract syntax model* to generate three separate diagram editors for the private, public and instance notation. The bottom level of Figure 11.10, M_2, shows the Quality Toys Inc. opened in the private editor, displaying all information about employees. To switch between notations a completely new editor has to be opened. Moreover, notations cannot be mixed as needed within one editor.

11.3.3 Summary R3: Multi-notation Modeling

The evaluation shows that in both approaches, the deep and non-deep, it is possible to define several notations for one model. Three main differences between the approaches exist in this respect. First, in the deep version a notation is defined within the deep model in form of visualizers attached to clabjects while in the EMF version the notation is defined in models which are external to the model defining the abstract syntax. The external storage of the graphical shapes can promote reuse by providing symbol libraries which can be reused in other languages. However, current lack of these libraries, given the maturity of the GMF framework, indicates that there is not a big demand for such reuse in practice. The other models used in GMF such as the mapping model, tooling model and generator model are so specific that there is little potential for their reuse. The clear disadvantage of the external storage of concrete syntax definition models lies in the domain of model evolution as previously outlined. Each time the abstract syntax evolves, the concrete syntax has to be co-evolved through some kind of tooling. In a deep model all concrete syntax information is directly coupled to the abstract syntax via pointers. Hence, this approach is more robust against evolution scenarios such as renaming or movement of attributes.

Second, in the GMF version, notations cannot be mixed and matched on-the-fly as needed. A user has to decide in which notation the model is viewed before it is opened. If a modeler decides to view a part of a model in a different notation, the diagram has to be opened in a different editor and the user has to navigate to the part viewed in the alternative notation. In GMF, a dialog which allows a modeler to select a notation, similar to the one present in Sirius,

has to be developed by hand to support this. These limitations have a high impact on the symbiotic language scenario [17] in which one notation enriches another. A modeling novice for example could switch parts of a model to the UML instance specification diagram to capture the meaning of graphical shapes in a concrete syntax. An example of this is shown in the deep version of the notation definition example where Steve is switched to the instance specification like LML visualization to capture information in a more explicit way (e.g. values are named through variables).

The third and final limitation is the lack of support for defining user-defined syntax at any classification level in the non-deep EMF version. In the example, this leads to the problem that the projectName attribute defined for ProjectLead at level M_1 (cf. Figure 11.8) cannot be included in the concrete syntax. In the deep modeling version, in contrast, this can be covered by defining a visualizer specifically for ProjectLead at O_1 in the same way as at O_0.

The evaluation can for example be extended to XText for defining and using textual notations. Again, the definition of the textual notation is separated from the abstract syntax in the EMF model and one different model for each notation definition has to be created. This raises the same model evolution issues as with the GMF solution. Also, the notation used to display a model in an XText editor cannot be switched on-the-fly. A new model editor displaying the alternative notation has to be opened instead, which prevents the mixing of notations as needed. Xtext's two-level technology only allows concrete syntax to be defined on the highest level, like GMF. This makes it impossible to define concrete syntaxes on O_1, e.g. to include the projectName attribute of ProjectLead in the textual concrete syntax.

11.4 R4: Multi-format Modeling

The multi-format modeling requirement is evaluated on four formats for editing the company structure modeling language — the diagram, table, text and form-based formats. One notation is defined in each format although in general it is possible to define more.

11.4.1 Deep Multi-format Modeling

Figure 11.11 shows how deep multi-format modeling is realized. Basically, one visualizer is defined for each format and is attached to the clabjects (diagram — D, text — Te, table — Ta and form — F). For space reasons, the O_1 level is elided and only the format-specific visualizers for DepartmentType are shown.

The diagrammatic visualizer specifies DepartmentType instances to be visualized as rectangles in the public and private notations. They display their name in the top of the rectangle and their location below the name. The compartment displaying the employees is indicated through the gray dashed line and occupies all free space in the department shape below the name and location. The textual visualizer defined for the public and private notation first prints the name of the department followed by curly brackets in which the employees working in the department are contained. The form visualizer for the public and private notation displays two rows with labels and text boxes. The first row displays the name of the department and the second displays the location. The employees are displayed in a list box at the bottom of the form.

In contrast to the previous three visualizers, the table visualizer for EmployeeType is shown in Figure 11.11 because when selecting a department for visualization the content of the department is displayed using the content's visualizer. The DepartmentType's visualizer only configures options such as whether to view the linguistic or ontological type column and what to display in the breadcrumb. Hence, it is hidden in favor of the EmployeeType's visualizer which describes what is actually displayed in the table. In the table each EmployeeType instance is displayed in its own row. The first column displays the employee's name, the second the salary and the third the expertise. This visualizer is only defined for the private notation because it contains sensitive information.

Having these visualizers defined, a modeler can invoke the different format-specific editors as needed at any level. In the example in Figure 11.11 the O_2 level is displayed in all four defined formats, the diagrammatic format is at the top left, the tabular format is at the bottom left, the textual is at the bottom right and the form-based format at the top right. A modeler using Melanee can actually align the different format-specific editors as shown in the figure. When using the format-specific editors, they are seamlessly synchronized with

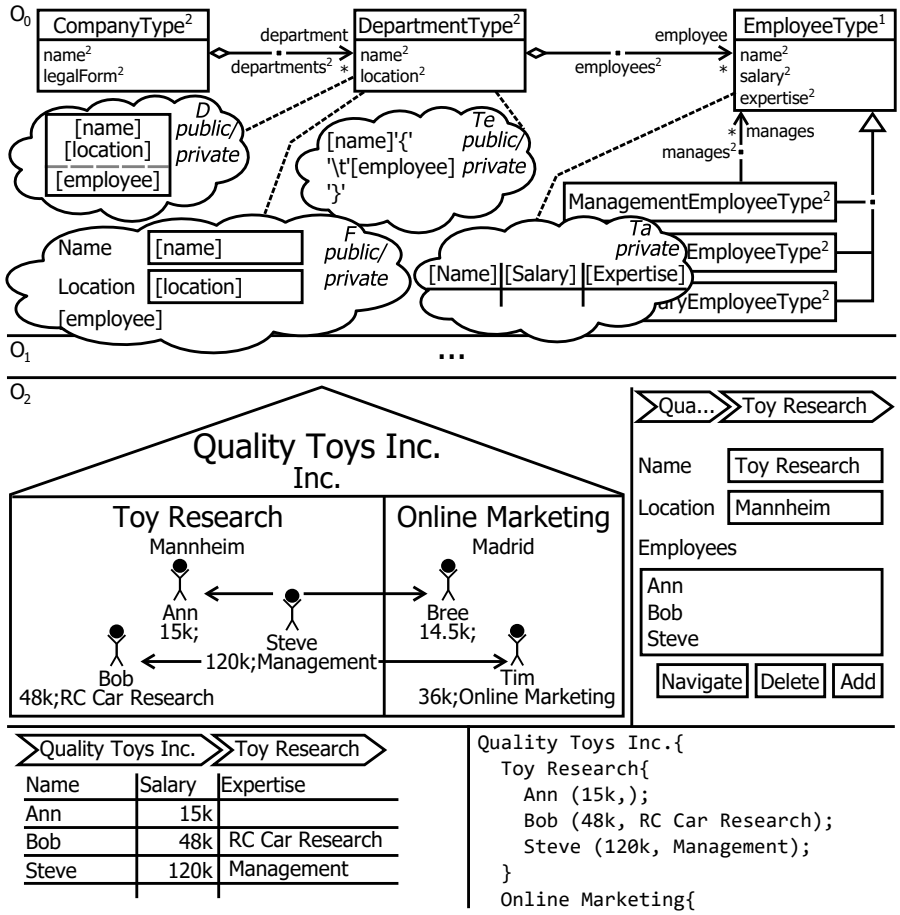


Figure 11.11: Deep multi-format modeling example.

each other and have equal importance. Changes in any format are immediately reflected in all other format-specific editors and do not impact any other format negatively. Although the example shows multi-format editing at the O_2 level, it can be used at the O_1 and O_0 levels as well.

11.4.2 Non-deep Multi-format Modeling

Multi-format modeling in EMF-based tools requires a distinct technology for each format. As previously shown, GMF was chosen for the diagrammatic format, while the textual format can be realized in XText [70] and the tabular

and form-based formats can be realized in EMF Parsley [191]. After defining the concrete syntax in each technology (i.e. format) using distinct formalisms, the editors have to be integrated with each other so that a modeler can invoke the different formats and work in them simultaneously. The most complex task when combining the different formats is to integrate the XText editor, because unlike the other editors it is a non-projectional editor. To integrate the XText editor a bidirectional model merging mechanism has to be implemented. The text displayed in the textual editor has to be updated in a way that formatting is preserved when one of the other formats commits changes to the edited model. When changing text in the textual model editor the parsed model has to be merged with the model underlying all other format-specific editors. Of special interest here is the diagrammatic GMF editor which stores layout meta data for the edited model elements. Hence, the merging of the parser generated model edited in XText with the model underlying all other formats must be implemented in a meta data preserving way.

The arrangement of this technology stack in context of the EMF version of the model is shown in Figure 11.12. The boxes at M_2 represent the different artifacts needed for the multi-format editing environment definition. These are the Abstract Syntax definition which shows an excerpt of the company structure modeling language from the EMF version in Figure 11.2, and the concrete syntax definitions in GMF, Parsley and XText connected to the Abstract Syntax by the dashed lines. Three different formalisms are used to define concrete syntax. GMF uses four EMF models as explained earlier, Parsley uses a dedicated, textual domain-specific language and XText follows a grammar-based approach popularized by tools such as ANTLR [189]. The editors created using these three isolated technologies are then integrated using Integration Code. To integrate GMF and XText, for example, the *resource* used by the GMF editor that is responsible for saving and loading models has to be replaced by a *resource* compatible to XText [64].

The resulting multi-format editing environment is shown in M_1. The three editors are aligned in the same way as in the deep modeling version. The difference between the two versions is that the modeling environment can only display M_1 level content and thus the editors are limited to this level. The editors, however, can be aligned as shown in the figure.

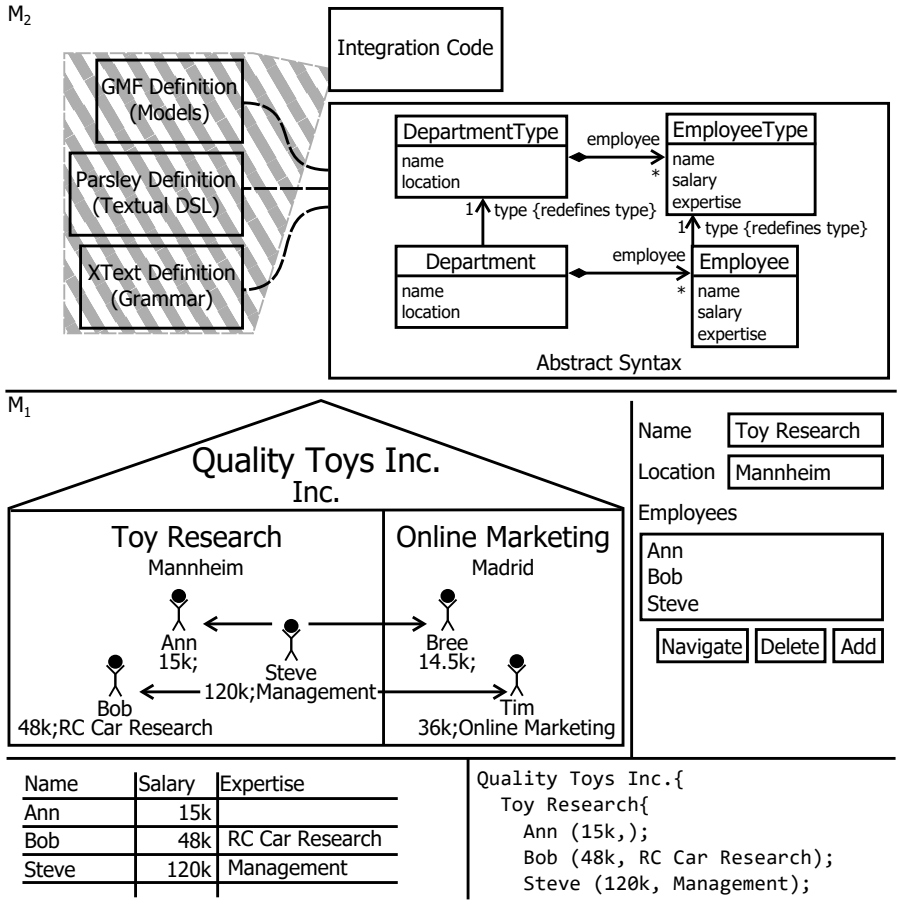


Figure 11.12: Non-deep multi-format modeling example.

11.4.3 Summary R4: Multi-format Modeling

In non-deep modeling three different technologies are used to define notations for the different formats while in deep modeling the concept of visualizers is used throughout all formats. Moreover, different non-deep modeling technologies use different paradigms for notation definition, e.g. textual grammar definition, tree-based models and textual models expressed in domain-specific languages. In deep modeling, however, the paradigm is the same across all formats — that is tree-based modeling. All format-specific visualizers share

the concepts of attributes, connections and expressions which are visualized in the target notation. The names are just slightly differing for the different formats to make them more format-specific, e.g. *AttributeLabel* in the diagrammatic format and *AttributeColumn* in the tabular format. In addition the languages share the concepts of visual grouping and layout, e.g. *Rectangle* and *TableLayout* for the diagrammatic format and *Group* together with its default *TableLayout* for the form-based format. In the non-deep EMF version, however, three distinct technologies with different terminology are used. For example, while GMF uses labels to display attributes this concept is not found anywhere in XText which uses only terminology originating from grammar definition languages such as EBNF.

From an integration point of view no additional work is needed to integrate the different format-specific editors in Melanee because its deep modeling approach is designed to deliver multi-format editing functionality out-of-the-box. However, as mentioned earlier, glue code has to be written to integrate the GMF and XText editor with each other. Moreover, since XText is based on free text parsing while the other editors are based on projectional editing, non-trivial merging problems arise such as how to proceed if a modeler saves a piece of text that does not conform to the concrete syntax or how to update the textual editor after changes to the model so that the current formatting is preserved. The grammars are also limited to the capabilities of ANTLR 3, XText's underlying parser generation framework. In contrast, projectional, deep model editors are not limited by the parsing restrictions of a parser and no merging problems arise because the textual format shares the projectional editing paradigm with all other formats.

Supporting evolution of the abstract and concrete syntax models is also a challenge in the non-deep multi-format editing scenario. For each technology a dedicated model co-evolution support tool has to be deployed (or developed if not available). In deep modeling, the format-specific notation definitions are stored together with the modeling language definition and pointers are used to point to model content which automatically updates the models on name changes etc. In cases where this is not sufficient, the emendation service can be extended to support the evolution of visualization definitions across all formats thanks to the common formalism and vocabulary of them.

11.5 R5 & R6: Context-sensitive and Aspect-oriented Visualization

Context-sensitive and aspect-oriented visualization is evaluated here for the diagrammatic format only. The results, however, are generalizable across all formats of the deep modeling approach because they are all based on the same underlying concepts. In the non-deep EMF technology space, the format offering the best capabilities in this area is the diagrammatic format. In general, the context-sensitive and aspect-oriented capabilities vary significantly depending on the format and technology employed. This is elaborated further in the summary of this requirement evaluation.

To evaluate context-sensitive visualization, the background of ManagementEmployeeType instances is colored red in case they manage an EmployeeType with a higher salary than the ManagementEmployeeType instance itself has. Aspect-oriented concrete syntax definition is demonstrated on the example of Researcher. Researchers shall be represented by a stickman wearing a square academic hat.

11.5.1 Deep Context-sensitive and Aspect-oriented Visualization

In deep modeling, context-sensitive visualization is realized by calculating values driving a visualizer through expressions defined in a deep constraint language. For this purpose, all visualizer elements in all formats can have a model element attached which defines the visualizer attribute to set and the expression to calculate its value. Whenever a change occurs to the abstract syntax model representation, the visualizations are updated to reflect the expression outcome defined on the visualizers.

An example for an expression defining the background color of a visualizer model element is shown for ManagementEmployeeType in Figure 11.13. The expression attached to the shape's background via a dashed line navigates to all managed employees and checks if one exists with a higher salary than that of the manager. If any employee has a higher salary than the manager red is returned by the query, otherwise white is returned. When this calculated

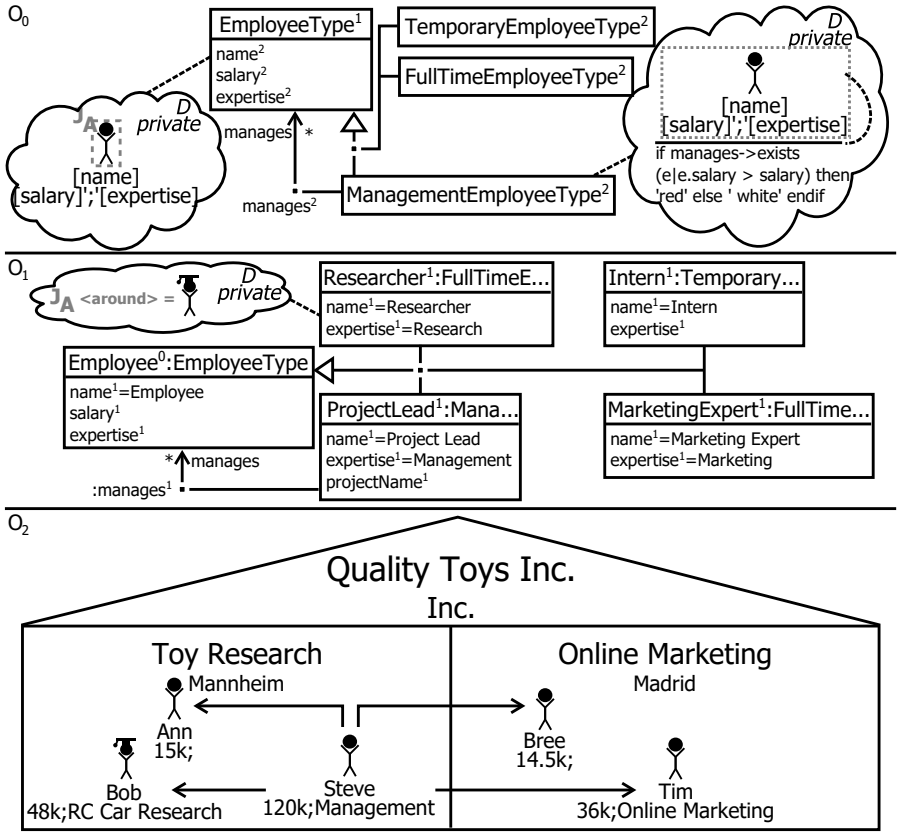


Figure 11.13: Deep context-sensitive and aspect-oriented visualization.

visualizer value is applied to Steve, a deep instance of ManagementEmployeeType, its background color becomes white because all employees managed by Steve have a salary lower than Steve's.

The visualization of Researchers as a stickman wearing a square academic hat is realized through the application of the approach's deep, aspect-oriented concrete syntax definition capabilities. The visualizer of EmployeeType defines a join point named J_A for the stickman symbol. Researcher replaces this stickman by a stickman wearing a square academic hat by defining an aspect of kind around for join point J_A. The visualization of Bob, a Researcher, shows the result of the aspect-oriented concrete syntax definition. Bob is visualized as a stickman wearing a square academic hat while the others are all visualized by plain stickmen.

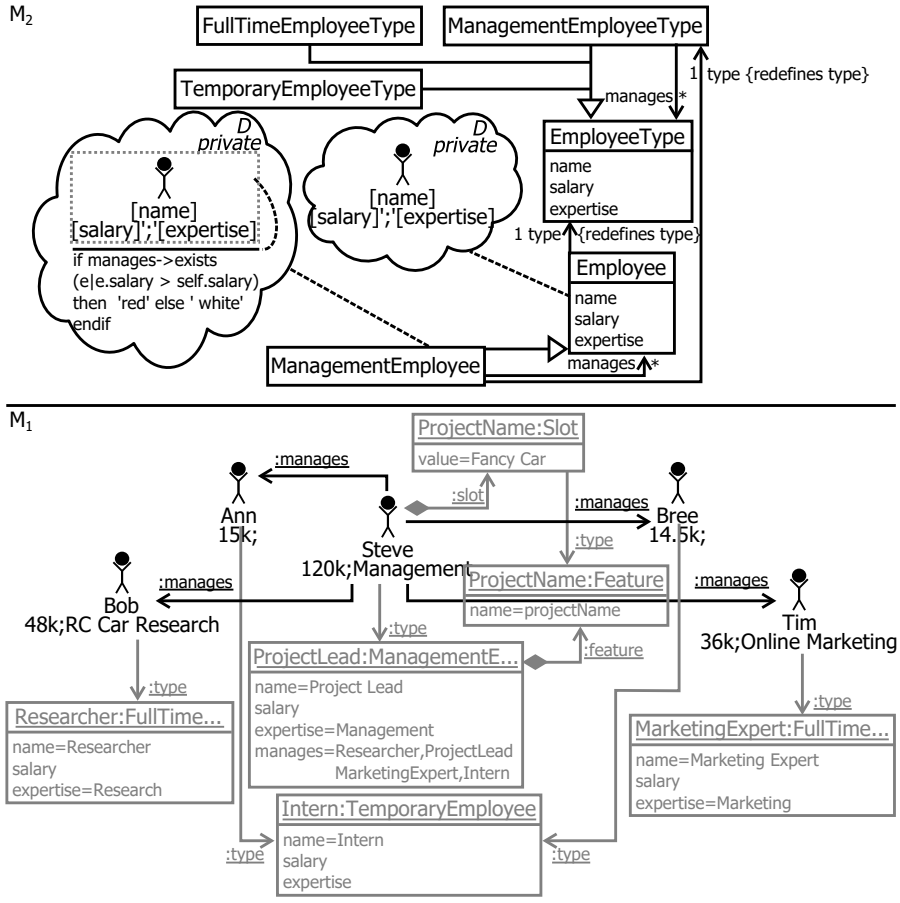


Figure 11.14: Non-deep context-sensitive and aspect-oriented visualization.

11.5.2 Non-deep, Context-sensitive and Aspect-oriented Visualization

The use of the non-deep version of the company structure modeling language, featuring context-sensitivity and aspect-oriented visualization, to model an actual company is shown in Figure 11.14. At the M_1 level one would typically only view the most concrete instances of the company to be modeled which here are the different employees actually working in the company. To have the same information present as in the deep model the types of the employees

are added to M_1 in gray color. The visualization is attached via clouds to classes in M_2. The clouds combine the *graphical definition model* and *mapping model* into one entity. To realize the context-sensitive visualization of ManagementEmployeeType instances more than one classification level away, a graphical visualization is defined for ManagementEmployee representing these instances. This context-sensitive diagrammatic visualization definition is supported by GMF through the *Pin* concept and constraints in the graphical mapping model. In this evaluation, Pins are used to set the background color of a shape depending on the outcome of an expression. This is again indicated through an expression connected via a dashed line to the shape's background as in the previous deep version of the example. Steve at M_1 is an instance of ManagementEmployee and earns more than all of the employees managed by Steve. Hence, its background color is white.

In this example the definition of the visualization works on instances of instances of ManagementEmployeeType because a dedicated subclass for ManagementEmployees is added to the Employee metaclass representing instances of ManagementEmployeeType two levels away. To specify a specific visualization of other employees e.g. Researchers an additional metaclass would have to be added as subclass of Employee to M_2 and a visualization would have to be provided for this newly introduced metaclass.

The aspect-oriented concrete syntax definition feature is not available in GMF and graphical definitions cannot be distributed alongside classification and inheritance hierarchies. Thus, an additional, fully-defined graphical visualizer has to be created for Researcher which applies the stickman wearing a square academic hat visualization to its instances. In EMF-based tools, however, a concrete syntax can only be defined on the M_2 level and Researcher resides at the M_1 level making it impossible to define a visualization for Researcher. As discussed earlier, the only way to achieve a specialized visualization for Researchers is to add a Researcher metaclass as subclass of Employee to M_2, retype all existing (M_1-)Researcher instances to this new M_2-type and define a concrete syntax for this M_2-type. This, however, conflicts with the premise that it should be possible to add new types to the M_1 level dynamically at run-time, because no specific visualizations can be defined for these.

11.5.3 Summary R5 & R6: Context-sensitive and Aspect-oriented Visualization

Context-sensitive visualization is available in deep modeling for all formats. Here the diagrammatic format is compared to GMF which supports this feature through the concept of pins and constraints in the mapping model. In other formats this feature is not present in the concrete syntax definition formalism. In XText, for example, java interfaces have to be implemented to achieve context-sensitivity to a certain degree. Furthermore, context-sensitive concrete syntax definition is limited to the M_2 level in EMF. Although it is possible to introduce new types at the M_1 level by applying the presented patterns for simulating deep modeling, it is not possible to define any specialized visualization including context-sensitivity on these types due to the lack of notation definition capabilities on M_1 model elements.

Aspect-oriented concrete syntax definition as demonstrated in the deep approach is not available in any of the concrete syntax definition technologies considered in the evaluation (i.e. GMF, XText, EMF Parsley). Hence, for each customization of a type's concrete syntax through subclasses and instances a fully defined visualizer has to be created at the type where the modification takes place. This visualizer duplicates all information of the original visualizer and adds its customizations. Moreover, in the evaluation, it was not possible to apply aspect-oriented concrete syntax definition along the classification hierarchy due to the lack of notation definition capabilities on M_1 model elements. Hence, it would not even be possible to simulate the capabilities of the deep model version by duplicating and customizing the notation at the M_1 type.

The evaluation also shows that the limitations of the EMF version and its accompanying concrete syntax definition technologies introduce additional accidental complexity in the presented example. First, accidental complexity is added to the abstract syntax because for each EmployeeType instance at M_1 which defines a new visualization, a new subtype of Employee has to be added so that instances of the newly defined types at M_1 are visualized in the corresponding visualization. This also contradicts the premise of dynamically adding new types at modeling time because M_2 and its concrete syntax is

ID	Modification Operation
C1	Background of departments is red if they employ over 50% managers.
C2	A department displays the average salary.
C3	A project lead displays "ZZZ" if more than 33% of the managed employees are interns.
C4	Employees have potency zero at the O_2 level.

Table 11.6: Set of constraints for evaluating deep constraint language support.

defined in a different modeling environment than M_1 and changes have to be propagated between the two modeling environments.

Second, accidental complexity is added to the concrete syntax definition. Fully specifying a new visualizer for each visualization customization significantly increases the number of model elements participating in the concrete syntax definition. A new visualizer has to be defined when the modification is as small as changing a visualization's icon which is done in this evaluation. This accidental complexity also tremendously increases the effort needed to evolve a concrete syntax, because all changes made to the general part of a concrete syntax must be transported to all customized visualizers which duplicate the general part of the concrete syntax for the pure purpose of visualization customization.

11.6 R7: Constraint Languages for Deep Visualization

The constraint language used to support deep visualization in Melanee is a deep OCL dialect [124] while GMF uses an implementation of OCL as specified by the OMG [184]. Hence, in this section it is evaluated how well the deep version of OCL and the EMF OCL implementation support the visualization of models. The constraints are defined in the context of the deep model displayed in Figure 11.1 and the EMF model displayed in Figure 11.2.

Textual descriptions of the constraints to be implemented during the evaluation are found in Table 11.6. These constraints cover the previously named features needed to support deep visualization which are: 1. deep constraint

application mode (*C1*, *C2*, *C3*, *C4*), 2. deep (re)classification operations (*C1*), 3. constraints using higher level navigations (*C2*), 4. definition on intermediate levels (*C3*), and 5. support for the OCA (*C4*) .

11.6.1 Deep Constraint Languages for Deep Visualization

The constraint for changing the background color of a department to red when more than 50% managers work within it is shown in Constraint 11.6.1. The constraint is defined as part of the DepartmentType's visualizer which is indicated by the context of the constraint (line 1). The constraint is then executed together with the visualization on all three levels of the deep model.

Constraint 11.6.1 (C1: Deep Department Background Color)**.**

```
1 context DepartmentType
2 let numberManagers:Integer=
3   employee→select(isDeepInstanceOf(ManagementEmployeeType))→size()
4 in
5 let numberNotManagers:Integer=
6   employee→reject(isDeepInstanceOf(ManagementEmployeeType))→size()
7 in
8 if ((numberManagers / numberNotManagers) > 1) then red
9   else white endif
```

To obtain the number of managers (numberManagers) working in a department the constraint navigates over the employee connection to all EmployeeTypes. Of these EmployeeTypes the ManagementEmployees are selected and the size of the set is determined (line 3). The number of non-managers working (numberNotManagers) in a department is determined using the same expression but rejecting all ManagementEmployees (line 6). Finally the constraint checks whether the ratio between numberManagers and numberNotManagers is greater than 1 and, hence, more managers are working in a department than other employees. If this is the case, red is returned as the color to use for the background, else white is returned.

Constraint 11.6.2 calculates the average salary in a department. It is therefore defined as part of the visualizer of DepartmentType indicated by the context

of the constraint. The constraint navigates to all employees and sums up their salary. Then the sum of all salaries is divided by the number of all employees working in the department.

Constraint 11.6.2 (C2: Deep Average Salary)**.**

```
1 context DepartmentType
2 employee.salary→sum() / employee→size()
```

To visualize a lazy project leader, an aspect is defined on ProjectLead at the O_1 level. The constraint used to control the application of the aspect is shown in Constraint 11.6.3. The constraint navigates to all managed employees via the manages connection provided by ManagementEmployeeType and selects all Interns of the managed employees. The number of managed interns is then divided by the number of all managed employees which is again retrieved via the manages connection provided by ManagementEmployeeType. Finally, the ratio between the managed interns and all managed employees is then calculated. If this ratio is higher than one third (here 0.34 is chosen) the aspect is applied because the constraint returns *true*.

Constraint 11.6.3 (C3: Deep Lazy Project Lead)**.**

```
1 context ProjectLead
2 _ManagementEmployeeType_.manages→select(isInstanceOf(Intern))
3                 →size() / _ManagementEmployeeType_manages→size() ≥ 0.34
```

When writing the constraint for the model as shown in the Figure 11.1 it is not necessary to use the navigation provided by ManagementEmployeeType. It is sufficient to use the manages connection defined by ProjectLead at O_1. This, however, would not support future connection diversification of ProjectLead. It is possible, for example, to extend the model in such a way that a ProjectLead has two manages connections with two different names. For instance, one could be for employees permanently managed by a project lead and one for employees only managed for a limited duration. In such a case it would be necessary to rewrite the constraint when not using the navigation provided at the type level.

Constraint 11.6.4 (C4: Deep Potency Zero Employees)**.**

```
1 context EmployeeType
2 self._l_.levelIndex = 2 implies self._l_potency = 0
```

Constraint 11.6.4 defined on EmployeeType forces all concrete employees working in a company to have a zero potency and thus to not be instantiable below level O_2. The constraint first navigates via the linguistic dimension and retrieves the index of the level in which the employee is located. If it is located at the second level (O_2) it checks whether the potency is zero by accessing the employee's linguistic potency trait.

11.6.2 Non-deep Constraint Languages for Deep Visualization

Constraint 11.6.5 is the non-deep version of the constraint which colors departments red when more than 50% of employees are managers. In contrast to the deep version the constraint is defined on Department as part of its GMF visualization definition and not on DepartmentType. This is necessary because the diagrammatic concrete syntax is defined on Departments, representing the O_2 instance facet, and not on DepartmentType, representing the O_1 type facet, due to the lack of a deep visualization and a deep constraint application strategy. Hence, visualizations and their accompanying constraints are not applicable to O_1 level content but only to O_2 level content when mapping the model elements from the EMF version to the ontological levels of the deep version.

Constraint 11.6.5 (C1: Non-deep Department Background Color)**.**

```
1 context Department
2 let numberManagers:Integer=
3   employee→select(isTypeOf(ManagementEmployee))→size()
4 in
5 let allEmployees:Integer=
6   employee→size()
7 in
8 if ((numberManagers / allEmployees) > 0.5) then red
9   else white endif
```

Similarly to the deep constraint previously defined in Constraint 11.6.1, first the number of ManagementEmployees (*let* expression in lines 2 - 4) is calculated and then the number of all Employees (*let* expression in lines 5 - 7) is calculated. Finally, the ratio of the numberManagers to allEmployees is calculated. If this

ratio is higher than 0.5 the background color is set to red, otherwise it is set to white.

Constraint 11.6.6 (C2: Non-deep Average Salary)**.**

```
1 context Department
2 employee.salary→sum() / employee→size()
```

Constraint 11.6.6 calculates the value displayed as the average salary for a department. The constraint is defined as part of a GMF expression label for Department, the instance facet of DepartmentType. In GMF, expression labels display the result of an OCL expression. As in the deep version, the constraint navigates to the salary of the employees working in the visualized department. The salary then is summed up and divided by the number of all employees working in a department.

Constraint 11.6.7 (C3: Non-deep Lazy Project Lead)**.**

```
1 context ManagementEmployee
2 type.name = 'Project Lead' implies manages→select(type.name = 'Intern')
3                                    →size() / manages→size() ≥ 0.34
```

The lazy project leader visualization cannot be defined on the ProjectLead class because it is located at the M_1 level which is not available for concrete syntax definition. Hence, Constraint 11.6.7 has to be defined on ManagementEmployee. Instances are then checked whether their type is named Project Lead. If this is the case they are checked if more than one third of the managed employees are interns. This second part of the constraint is equal to the deep version.

Constraint 11.6.8 (C4: Non-deep Potency Zero Employees)**.**

```
1 context Employee
2 – not supported
```

Constraint C4 cannot be expressed in the EMF-based, non-deep OCL version for two reasons. The first reason is that access to the linguistic dimension is not available in OCL as defined in the OMG Standard [184]. The EMF version of OCL, however, offers the workaround of casting any model element to the EMF base class *EObject* which provides reflective access to all model

elements. This allows linguistic attributes of model elements to be accessed. The second reason is that in the non-deep EMF version of the company modeling language the deepness of a class is not controlled via potency. Deepness is controlled via explicit modeling of types and their instances at the M_2 level. Thus, to raise the potency of EmployeeType an additional model element defining a type relation to Employee, the metaclass representing the instances of EmployeeType, has to be added to M_2.

11.6.3 Summary R7: Constraint Languages for Deep Visualization

Both the deep and non-deep versions of OCL are able to support the constraints defined in the evaluation, except for *C4* which is not supported in non-deep OCL due to the lack of the potency concept in EMF. While the deep version allows constraints to be defined at the most abstract level and then applied to all ontological levels, this is not possible in the non-deep EMF version. In the EMF version visualizations and constraints are always defined on the metaclasses representing the instance facet of types defined at M_1. If constraints shall be applied to types and instances residing at M_1, they have to be defined twice on M_2: once on the type metaclass and once on the instance metaclass of M_1 model elements.

Furthermore, the non-deep version of OCL lacks the option to define visualizations and constraints on model elements introduced at intermediate classification levels, e.g. ProjectLead at M_1. Hence, it is not possible to define constraint *C3* on ProjectLead. A workaround is to define the visualization and its constraint on the instance facet of the types specified at M_2 and check the name of the type referenced via the type reference on M_1 as application condition of the constraint. In the example this pattern is applied when defining *C3* on ManagementEmployee. By default, the constraint is applied to all instances of ManagementEmployee and thus to all instances of ManagementEmployeeType instances. To restrict the execution to instances of ProjectLead only, the typing ManagementEmployeeType instance is checked to see whether it designates the type Project Lead in its name attribute via the type reference between ManagementEmployeeType instances and their instances.

Connection diversification and navigation over connections introduced on ontological type levels other than O_0 and hence M_2 in the EMF version of the company structure modeling language is not supported in the non-deep OCL version. An example is a ProjectLead connected to managed employee types via more than one navigation name such as *managedFTEs* and *managedInterns* at M_1. The constraints defined at M_2 are blind to these newly defined navigation names at M_1 and only manages, defined at M_2, is available for navigation at this level. To realize a navigation via these identifiers defined at M_1 a navigation based on the instance names of the manages reference has to be applied. The deep version supports this scenario by being able to define the constraint at the type introducing new navigations independent of the classification level at which this type resides. In the example, the constraint would be directly defined on ProjectLead at O_1 and thus all navigations defined by this type are available for navigation in the constraint.

Access to the linguistic dimension is provided by EMF OCL and the deep OCL dialect of Melanee. While deep modeling offers native support for accessing the linguistic dimension of a model element, workarounds have to be applied in the EMF version. These include casting to the base class *EObject* of all EMF model content and then invoking EMF's reflective API. Constraint *C4*, however, could not be supported in EMF's OCL version due to the lack of the potency concept in EMF. It would be possible to raise the potency of a model element by changing the M_2 metamodel of a language and introducing a type for the new instance level. However, this is not possible in an EMF-based modeling tool without recompilation and deployment steps.

In the EMF modeling environment used for this evaluation, OCL constraints are not supported for all modeling formats. For example, XText for defining notations in the textual format does not allow these to be enriched with context-sensitivity through constraints in its concrete syntax definition formalism.

11.7 Evaluation Summary

The evaluation shows that it is possible to build a deep, multi-format, multi-notation modeling workbench with today's technology using the Eclipse plat-

form as the basis for the modeling environment, EMF for metamodel definition, GMF for diagrammatic editor definition, XText for textual editor definition and EMF Parsley for form and table editors. The evaluation also showed that this is only possible by applying workarounds introducing accidental complexity.

To achieve deepness of the company structure modeling language the patterns described by de Lara et al. [151] have to be applied. When applying only the patterns necessary to transfer the content modeled in the deep version to the EMF version, the number of model elements nearly doubled and significant accidental complexity was introduced. Applying further patterns giving modeling capabilities to the EMF version similar to those of the deep modeling version increased the complexity even further.

Also the lack of integrated concrete syntax modeling technology in EMF creates complexity. The different format-specific technologies for defining concrete syntax are not built with each other in mind. Thus isolated *island solutions* are created which have to be manually integrated with each other. A full, seamless integration of the different formats can only be achieved with significant development effort. Furthermore, the presented tools are built to support one notation at a time when working with a model in an editor. To support a multi-notation feature the editors supplied by the format-specific concrete syntax definition frameworks would have to be manually tailored. The high heterogeneity of the tools not only causes increased development effort related to concrete syntax development but also significant additional complexity. So for example in GMF three different mapping models had to be created to support modeling in three different notations due to the lack of multi-notation support out-of-the-box. Also the differing formalisms for defining concrete syntax introduce complexity because a modeling language engineer has to learn all of these formalisms in order to define concrete syntax or understand their definition.

The lack of context-sensitivity in all formats except the diagrammatic format, backed by GMF, and the lack of aspect orientation throughout all technologies for defining concrete syntax also introduces additional complexity. In formats such as XText, context sensitivity has to be implemented by enriching generated source code, thus breaking the model-driven development paradigm

and increasing the effort needed to maintain a developed language. The lack of aspect orientation leads to duplication of concrete syntax definitions and thus to increased accidental complexity. This duplication also makes language maintenance much more difficult.

In general, it can be observed throughout the evaluation that complexity is introduced whenever deepness has to be emulated due to the lack of inherent support for deepness in the modeling technologies accompanying EMF. Examples are the inability to define visualizations on M_1 model elements in any of the concrete syntax definition frameworks and missing support for defining constraints on types introduced at the M_1 level.

Even though the evaluation did not go into full details about each format but rather focused on the underlying concepts, it found the EMF environment to be deficient with respect to all the modeling environment requirements established in this thesis. The deep, multi-notation, multi-format approach presented in this thesis, supported by Melanee, in general performed much better than the EMF version in all areas of the evaluation and fulfills all modeling goals identified in the introduction chapter. The reason, of course, is that the Melanee framework was designed with all these requirements in mind in contrast to the EMF technologies which independently evolved for one specific purpose (e.g. textual DSL definition).

Chapter 12

Related Work

This chapter gives an overview of work related to the approach presented in this thesis. It considers multi-level and non multi-level modeling tools that provide state-of-the-art support for domain-specific language engineering and/or multi-level modeling. The following sections describe the most influential academic and commercial technologies available for multi-level modeling and for leveraging the different formats considered in this thesis. They first provide an introduction to several commercial and academic tools that are subsequently considered in detail. Tools that are either not big competitors in the market or not actively maintained are named but not closer described. The chapter closes with a tabular comparison of all the closely examined tools.

12.1 Diagrammatic Domain-specific Language Workbenches

In the area of diagrammatic language workbenches five tools are considered as related work. The Graphical Modeling Framework, Sirius and MetaEdit+ are three of the most widely distributed commercial tools, while AToMPM and the Generic Modeling Environment (GME) are two of the most influential academic tools. All of these tools focus exclusively on the diagrammatic format and are thus not regarded as *multi-format* except Sirius and MetaEdit+ which also support tables and trees but do not support the textual format.

A limited kind of *multi-notation* editing can be achieved in Sirius through its viewpoint and layer features and in MetaEdit+ through its layer feature. However, none of the diagrammatic language workbenches considered in this section supports *seamless*, *deep modeling* since they are inherently based on a single class/instance level pair. Moreover, none of the tools supports aspect-oriented, concrete syntax definition and provides a predefined notation for visualizing model instances out-of-the-box. Context sensitivity is supported by all of the tools except GME.

The tools Poseidon for DSL [88], Microsoft Modeling SDK for Visual Studio [169], XModeler [49], Visual Modeling and Transformation System [168, 230], TIGER [71, 222], Clooca [111], GMFGen [207], Intentional [115] and EuGENia [140] are not analyzed in detail due to their relatively limited user bases.

12.1.1 Graphical Modeling Framework Tooling

The Graphical Modeling Framework (GMF) [94] consists of two components — the Graphical Modeling Framework Tooling (GMF-T) and the Graphical Modeling Framework Runtime (GMF-R). GMF-T allows a language engineer to create graphical modeling workbenches by defining a set of models. These models are then translated into Java source code which uses the GMF-R offering an API for implementing graphical EMF model editors. The GMF-R API essentially is an extension of the Graphical Editing Framework (GEF). GEF is built on top of the Standard Widget Toolkit, Eclipse's UI Toolkit, and aims to support the simple creation of graphical model editors leveraging the Model View Controller Pattern (MVC). The extensions provided by GMF-R cover the basic functionality needed when creating GEF editors for models based on the Eclipse Modeling Framework (EMF).

In GMF-T, five models are involved in the definition of a diagrammatic model editor — the domain model, the graphical definition model, the tool definition model, the mapping model and the generator model. The graphical definition model offers shapes (e.g. rectangle, label) and layouts (e.g. table layout) to model the appearance of domain model elements. The domain model describes the model elements which are available in a language and how they relate to one another. The tool definition model describes the tools available for model element creation in the palette placed at the right-hand side of

the model editor. These three models are linked together using the mapping model. It defines mappings from domain elements to tools which create them and graphical definitions which visualize them. The mapping model can specify further information such as the default initialization of domain elements or constraints which are evaluated on the domain model. The mapping model is used to generate the generator model which configures the code generation process. This model includes information on how the generated plug-in is named, which dependencies are used etc. The processes for creating the generator model and the generated code can be customized by the user. The creation of the generator model is customized by using QVTo [185] templates, while XPand-based [69] Model-to-Text (M2T) transformation templates can provide aspects which customize the default code generation templates [197]. These two techniques allow a language engineer to enhance the GMF-T framework with functionality which is not provided out-of-the-box.

12.1.2 Sirius

At the time of writing, Sirius [43, 63, 211, 231] is the latest contribution to the Eclipse Platform in the area of domain-specific language definition. Like GMF-T, Sirius is built on GMF-R and can be used to build diagrammatic modeling workbenches using models. In contrast to GMF-T, these models are interpreted at run-time. This means that editors can be changed on-the-fly by changing the model that defines them. The tool focuses on providing editing capabilities using different viewpoints offering different editors which can be diagrammatic, tabular or tree-based. Other formats are possible but not implemented at the time of writing. These features could be used to emulate multi-format and multi-notation editing by opening a model in different viewpoints defining different notations and formats. This however, would still not allow modelers to mix notations freely in one editor as the notations are isolated from each other in distinct editors. Moreover, the textual format is not available for editing in Sirius out-of-the-box. XText can be manually integrated with the other Sirius formats but has some weaknesses because of the integration of the projectional Sirius editors with the non-projectional XText editors [195].

The different Sirius editors can offer different levels of granularity when

viewing a model. Using layers it is possible to hide parts of a tool palette, introduce new model elements or influence the current style (i.e. visual shape) of model elements. Editors, by default, do not show any model. They need to be populated by hand with the model elements they are intended to display. This can be done via drag-and-drop, a button in the tool palette or in other ways which the language engineer can freely define. Views on a model are by default synchronized with each other, but this can also be influenced by a language workbench engineer who can choose different modes of synchronization including no synchronization.

To define a language workbench one monolithic model, the View Point Specification Model (VPSM), is created which defines how the modeling workbench behaves. This model defines the viewpoints that are available in the final modeling workbench. The viewpoints contain the definitions of the different editors which are defined by the graphical shapes used to visualize model content, the tools that are available to manipulate the model and a mapping between the domain models and the graphical visualizations.

The graphical definition is created similar to the GMF-T by combining several predefined styles (i.e. shapes such as squares, triangles, diamonds etc.) to visualize domain elements. In the mapping definition, domain elements are mapped to their corresponding visualizations. The mappings to the domain model consist of two parts, the mandatory *domain element* to which the mapping applies and an optional *candidate expression* which allows selected model elements to which the mapping shall be applied to be further refined based on domain rules in a constraint language.

In the tool definition, actions available for model manipulation are defined. The tools are modeled using a small action language which allows constructs such as create instance, delete view, navigation, set value etc. to be defined. These actions can be further refined by the modeling language engineer using a constraint language. In cases where a constraint language is not sufficient, such as the parsing of input after the direct editing of a label, calls to functionality coded in Java can be made. Additionally, layers, toolbars, filters, and decorators can be defined for a model. A layer allows mappings, tools and graphical customization to be defined which are only available when the layer is selected by a modeler.

12.1.3 MetaEdit+

MetaEdit+ [166, 224] is, at the time of writing, the most successful commercial workbench for defining diagrammatic domain-specific languages. It focuses on the definition of diagrammatic languages and the generation of running programs from models created in these domain-specific languages. MetaEdit+ uses a four step approach to define domain-specific modeling workbenches: 1. Concepts, 2. Rules, 3. Symbols and 4. Generators.

Concepts (domain models) are either defined in a dialog box or graphical editor based on the preference of the domain-specific language engineer. A proprietary metamodeling language is used by MetaEdit+ which offers well known concepts like entities, attributes etc. Static semantics is defined using the *rule* editor. Templates are provided (e.g. maximal occurrence of a metamodel instance) which are parametrized to support the writing of rules. Rules can be checked and enforced directly when a model instance is created.

The *symbol* editor offers convenient *what you see is what you get* editing of metamodel element visualizations. Graphical shapes can change their visualization in a context-sensitive way depending on defined rules. A language engineer can define different layers of abstraction for one model, so that for example a detailed view can be shown to domain experts and a simplified view can be shown to business stakeholders. MetaEdit+ supports diagrammatic, matrix and table editors. Hence, it is also possible to define multiple notations and multiple formats for a model. Again, as with Sirius, the textual format is not available and notations cannot be freely mixed within one editor.

Generators allow models to be serialized to textual formats such as XML or a programming language for further processing. Either the predefined built-in generators can be tailored and reused or custom generators can be defined using the generator editor. Advanced tools exist to debug the generators, including step-by-step execution, breakpoints and tracing of text output to the original graphical model element.

A key feature of MetaEdit+ is its support for metamodel evolution. All changes which are made at the metamodel level are immediately reflected in the rules and generator definitions. In particular, existing models still conform to the evolved metamodel in contrast to modeling infrastructures like EMF. This conformance, however, is not achieved by applying model evolution operations

but declaring the old typing information as still valid [225]. This facilitates the long term evolution of models over several decades.

MetaEdit+ offers an API and interchange format (XML) for integration with external tools. A central repository is used to store all model content — concept definitions, rule definitions, symbol definitions, generator definitions and model instances. Collaborative teamwork is made possible by means of model sharing. Multiple modelers can work on different parts of a system at the same time without interfering each other. MetaEdit+ takes care of synchronization and conflict management during collaborative work.

12.1.4 AToMPM

AToMPM [77, 163, 219], the successor of AToM^3 [152], is an open and extensible web-based multi-paradigm modeling tool for modeling in the cloud. AToMPM runs in every SVG compliant web browser. It focuses on the definition of abstract syntax, multiple concrete syntaxes which can be mapped to an abstract syntax, transformations and collaborative modeling. The AToMPM language workbench itself is an interpretation of models created in AToMPM. Hence, full customization of the modeling environment through the usage of AToMPM models is supported.

The default language for defining abstract syntax is a simplified UML modeling language which can be replaced by any metamodeling language as needed. Static semantics is defined in a textual DSL or JavaScript. The concrete syntax is defined and mapped to the abstract syntax in so-called icon models. These are defined through a DSL tailored for defining SVG-based graphics. In this language all SVG properties are available to create graphical shapes (icons). Context-sensitive concrete syntax definition is also supported. It is possible to define multiple concrete syntaxes for one abstract syntax. A language user can exchange the concrete syntax that is used for modeling on-the-fly with the click of a button. In addition to switching the concrete syntax, the model editor also supports model elements in the editor to be hidden so that the content displayed in the editor is limited to the information needed for the needs of a specific modeler.

A unique feature of AToMPM compared to the other tools presented here is collaborative modeling by either screen sharing or model sharing. In the

first scenario multiple modelers work simultaneously on the same view of the model. In the second scenario multiple modelers work on different views of the model which can for example have different concrete syntax. Changes by one modeler in one view are synchronized to the views of all other modelers.

Transformations are used, amongst other things, to simulate, animate and analyze models. Additionally, they are used to translate from one abstract syntax to another. These model transformations are defined in a rule-based transformation language generated from the domain-specific input and output modeling languages [216]. This enables a modeler to use the domain-specific input and output languages to define the preconditions for a rule to be applied, optional negative application conditions (patterns to which the rule is not applied) and the post condition (result of transformation rule). The transformation rules are scheduled using the Motif [217] language. Furthermore, the transformation language included out-of-the-box can be extended with new concepts or fully replaced. This is achieved by specifying a higher-order transformation which translates the newly introduced concepts into the transformation language shipped with AToMPM (T-Core [218]). Using a graphical debugger, transformations can be paused, executed step-by-step etc.

Complex editing operations can be performed via scripts that are typed into the command line at the top of the editor. An example of such an operation is to create five new places in a petri net.

12.1.5 Generic Modeling Environment

The Generic Modeling Environment (GME) [90, 154, 155] is a configurable domain-specific modeling environment. A model containing the abstract syntax, static semantics and concrete syntax is used to configure the environment to the desired modeling paradigm for the modeling workbench to be developed. The GME language workbench itself is built using such models, and hence GME is defined in GME.

In GME domain-specific languages are defined through so-called modeling paradigms and aspects. A modeling paradigm defines the domain-specific modeling language (i.e. abstract syntax, concrete syntax and semantics), whereas an aspect defines a special view on a model (e.g. which model elements are visible or can be created). GME uses a modeling language that consists of

models, *atoms*, *connections* and *references*. A *model* is a container that can contain other elements such as *atoms* or *connections*. An example of a *model* could be an actor that contains the steps it is executing. Every GME editor has one root *model*. *Atoms* are the most atomic constructs in a language which cannot contain any other constructs. Examples would be gateways in a process modeling language such as *XOR* or *AND*. *Connections* are model elements themselves which can contain *attributes* and can connect model elements such as *models* and *atoms*. They are restricted to connecting model elements within the same container only. *References*, which are essentially like pointers in a programming language, can be used to overcome this restriction, but they cannot have *attributes*. The language engineer uses a metamodeling language close to the UML to define modeling paradigms. The created classes indicate which concepts in GME they are related to by applying stereotypes. Static semantics can be defined using an OCL like constraint language. The visualization of a model is also defined in the paradigm metamodel by assigning icons (i.e. bitmap graphics) to atoms and models. Additionally, graphical properties such as the color and kind of lines (e.g. dashed) or the color of model element backgrounds can be set. Moreover, it is possible to define which metamodel elements are visible in an *aspect* by modeling these together with the metamodel of the modeling paradigm.

GME can be extended using the COM programing model. An API to access and manipulate models programmatically is provided, as well as an event mechanism to which plug-ins can subscribe. The storage format of GME is either a proprietary binary format or an XML-based format.

12.2 Textual Domain-specific Language Workbenches

This section covers the major commercial, textual, domain-specific language workbenches, – XText [70] as the representative for grammar and parser-based language workbenches and JetBrains MPS [41] as the representative for grammar and projection-based language workbenches – as well as the academic EMFText [103] as the representative of parser-based language workbenches

which annotate metamodels with concrete syntax information.

Of these, only JetBrains MPS supports multi-format editing out-of-the-box. In MPS, for example, a table or mathematical formula editor can be hosted within the textual editor. It is also the only projectional, textual language workbench discussed here. The language for defining concrete syntaxes is similar to Melanee's which mainly relies on *literals* and *values*. EMFText and MPS both provide a predefined textual syntax. Multi-notation editing is not possible in any of the tools. Separate editors are defined for each notation and files need to be opened with the notation-specific editors. None of the tools supports seamless deep modeling since they are inherently based on a single class/instance level pair. Moreover, none of these tools supports aspect-oriented concrete syntax definition or context-sensitive syntaxes.

Many other textual domain-specific language workbenches exist on the market at the time of writing, such as TCS [121, 120], which was used to create the ATL [122] editors, Spoofax [125], Monticore [142], Whole Platform [212] and Meta Model Syntactic Sheets [76]. However we do not analyze these in detail here due to their relatively limited user bases.

12.2.1 XText

XText [70, 240] is an Eclipse-based textual language workbench for creating textual domain-specific languages. These languages are supported through features such as syntax highlighting, code completion, code generators etc. Editors for Eclipse, IntelliJ IDEA [144] and web browsers can be generated.

XText supports two modes for defining a domain-specific language, one is to start with a grammar definition and infer the EMF-based domain model from this grammar and the other is to start with an EMF-based domain model and import it into the XText grammar definition. Both modes can be mixed as needed within one grammar. One part of a grammar can be, for example, used to infer an Ecore model and the other part can reuse an existing Ecore model. The grammar definition language is very similar to standard grammar definition languages such as ANTLR or EBNF. Since the underlying parser generation framework of XText is ANTLR3 it is limited to grammars which are parsable by an ANTLR3 generated parser. This excludes left recursive grammars, for example.

The generated editor can be customized through the MWE2 [171] based language generator and Google Guice [228] based dependency injection. The MWE2 customization is used to configure code generation properties such as the file extension of the model instances or which platform is supported (e.g. eclipse or web). Dependency injection is used to extend the functionality of XText through such things as custom value converters which convert the parsed text into data conforming to a data type. An example for the application of a value converter is to parse cardinalities to integer values in situations where an infinite upper bound is represented by the non-integer *star* (*) literal in the textual language but internally represented by an integer value of *-1* as e.g. in EMF Ecore.

12.2.2 JetBrains MPS

JetBrains MPS [41, 232, 234] is an IntelliJ IDEA based language workbench for the development of textual, projective programming environments. Through its projective editing capabilities it is possible to host other formats such as table or mathematical formula editors within the created text editors.

A user creates a language in MPS by defining a structure model (abstract syntax), editor model, constraints, behavior, a type system and generators. The structure is modeled by a combination of an EMF-like tree editor and text for specifying the properties of metamodel elements. In contrast, Eclipse EMF uses property tables for this task. The metamodel elements available in MPS are *Concept* (i.e. metaclass), *Enum Data Type*, *Constrained Data Type*, *Primitive Data Type Declaration* and *Interface Concept*. *Interface Concepts* define properties, containment references and references to be reused by *concepts*. The defined structure can be edited in a predefined language shipped with MPS. To replace this predefined language with a user-defined language the editor model is employed. For each concept from the structure a *Concept Editor* defining the concrete syntax is created in a textual language. *Editor Components* representing concrete syntax definition snippets to be reused by *Concept Editors* can also be defined. The concrete syntax definition language consists of *layout managers*, for laying out the text, *labels* showing static strings and *values* displaying values from the underlying model.

Constraints are defined in the constraint model in a Java-like constraint

DSL and can be linked to properties or references. Code generation is defined in the generator model via a rule-based modeling language which maps model elements to templates. This concept works in a similar way to model to text transformation languages such as Acceleo [2], an implementation of the MOFM2T [180] OMG standard. Behaviors, defined in a Java-like textual syntax, define how a concept behaves in the editor. An example is a *for-loop* making its *loop-variable* available to all statements in it (scoping), or the definition of default values on instantiation. Type systems, defined in a textual DSL, define rules on types which are used for validation. For instance, it is possible to define that a *string* cannot have an *integer* assigned to it.

12.2.3 EMFText

EMFText [74, 103] is an EMF-based tool for annotating Ecore metamodels with textual concrete syntax definitions. Furthermore, it supports the generation of default syntaxes (e.g. HUTN) for Ecore metamodels which can then be tailored towards the desired user-defined language. An EMFText definition always consists of two artifacts, the Ecore model for which the language is defined and the file containing the annotations to the Ecore model in a textual language which looks similar to grammar specification languages like ANTLR or EBNF. Like in XText, fully functional Eclipse editors including syntax highlighting, code completion etc. are generated out of the Ecore metamodel and the concrete syntax definition. The UI independent code of EMFText does not have any dependencies to Eclipse or EMFText and can thus be run on any platform supporting Java.

The generated editor can be customized by either *overriding generated artifacts*, *overriding meta information classes* or *using generated extension points* [73]. For parts which are independent of the grammar the authors of EMFText recommend to *override generated artifacts* which involves manually changing generated code and directing the code generator to not replace the modifications. They point out, however, that in future EMFText versions a revision of the modified editor source code may be necessary. The approach of *overriding meta information classes* involves changing the factories generated by EMFText to return subclasses of the originally generated classes. The authors recommend this for customization of concrete syntax relevant code.

The extension points offered by EMFText provide customization capabilities for model loading and making extensions to the parser.

12.3 Form-based and Tabular Domain-specific Language Workbenches

At the time of writing no language workbench exclusively focused on creating form or table-based languages exists. However, such formats are covered by more general tools that address other formats as well. Tables, for example, are supported in Sirius although its focus is on diagrammatic language engineering. Forms and tables are covered by EMF Forms [72] as part of the EMF Client Platform [193] and by EMF Parsley [191]. EMF Parsley uses textual models while Sirius and EMF Forms use tree-based models for defining forms and tables. All three define the concrete syntax in a separate model from the abstract syntax. Of the three tools, Sirius offers the most customization possibilities for tablular, domain-specific language definition beyond the mere mapping of columns to metamodel element attributes as supported in EMF Forms and EMF Parsley. However, none of these tools offer explicit multi-format or multi-notation support (except Sirius partially through its viewpoint and layers feature) and they do not support deep, seamless editing since they are inherently based on a single class/instance level pair. Furthermore, no predefined tabular or form-based language is provided by any of the tools and they do not support aspect-oriented concrete syntax definition.

12.4 Multi-level Modeling Tools

The main multi-level modeling tool competitors to Melanee are MetaDepth and DPF. Of the two, the tool which offers the most similar capabilities to Melanee is MetaDepth. Like Melanee it is based on the orthogonal classification architecture and provides a textual predefined notation for deep models, a deep constraint language and support for defining textual user-defined languages. DPF is a diagrammatic deep modeling tool which allows modelers to define diagrammatic user-defined languages and provides a predefined dia-

grammatic language out-of-the-box. However, it does not provide a constraint language. Both tools lack support for multi-notation/multi-format, context-sensitive editing, and aspect-oriented concrete syntax definition. Seamless modeling is supported without an emendation service and thus the complexity of changes has to be handled manually by a modeler.

At the time of writing there are many other multi-level modeling tools such as the Open MetaModeling Environment [236, 237], Cross Layer Modeler [57], Modelverse [227], Nivel [4], MultCore [161], XModeler [48, 49, 83, 104], Xome [91] and Deep Java [146] . However, we do not analyze these in detail here due to their lack of maintenance, or relatively limited user base.

12.4.1 MetaDepth

MetaDepth [50, 51, 54, 167] aims at supporting textual, deep modeling based on the orthogonal classification architecture. Hence, it follows the same basic deep modeling approach as the one supported by Melanee. The target audience of MetaDepth is developers who want to use a textual syntax to define executable deep models. For this purpose MetaDepth is built on the epsilon framework [187] which provides a textual HUTN [178] like notation for model definition and an OCL-based transformation [138], code generation [201], action [137], validation [139] and query language. Using these ingredients MetaDepth provides powerful features for creating deep models that can be executed via an action script written in the Epsilon Object Language (EOL).

An in depth comparison of MetaDepth and Melanee is provided in [12]. This comparison shows that most of differences between the languages underpinning the tools originate for their different foci. While Melanee primarily targets language engineers aiming at defining user-defined languages in any format, MetaDepth focuses on supporting programmers aiming at defining deep models that can be executed or used in deep action, transformation and generation languages. This influences the classification semantics of MetaDepth, which are more strictly focused on creating models which are easier to program against with as little ambiguity as possible. Example features are *leap potency* and *deep references*.

For concrete syntax definition, MetaDepth uses a template based ap-

proach in which every clabject has a template attached describing its concrete syntax. The visualization search algorithm from [76] (on which Melanee is also based) is used to find templates for clabjects in their classification and inheritance trees should they not have templates themselves. The definition of concrete syntax templates mainly relies on *literals* to define static, non-changeable text and *values* showing the values of clabject attributes. In MetaDepth a potency is assigned to each template to define how many levels below a concrete syntax definition a template is applied. This differs from Melanee which applies a template as long as no new template is defined in the inheritance or classification hierarchy.

The deep constraint language of MetaDepth supports navigation between the linguistic and ontological dimensions and navigation via reference names defined on higher ontological levels. Hence, the Melanee and MetaDepth constraint languages are very similar. One of the few differences is the default level at which constraints are evaluated. In MetaDepth constraints are evaluated on the level defined by their potency, whereas for the deep visualization scenario, Melanee constraints are by default applied across all classification levels. In addition to plain constraint definition, MetaDepth also provides capabilities to check whether a constraint is satisfiable or not [52, 96] by using the USE Validator model finder [145]. This feature is not present in Melanee.

12.4.2 Diagram Predicate Framework

DPF [81, 136, 148, 202] is a diagrammatic, deep modeling tool with capabilities for defining diagrammatic user-defined languages. DPF is available as a web tool (*Web DPF*) and Eclipse plug-in. The version of DPF described here is the Eclipse plug-in version which supports diagrammatic user-defined languages (unlike the web-based DPF version). DPF adopts a formal approach to deep modeling like Nivel [4]. It is based on the *generalized sketches* formalism [59] and *category theory* [34]. While Nivel has not been implemented, however, two implementations of DPF exist as formerly described. From an implementation point of view the tool is based on the Graphical Editing Framework and the Eclipse Modeling Framework like Melanee.

Since DPF assumes a linear organization of linguistic and ontological levels it is not based on the orthogonal classification architecture like Melanee and

MetaDepth. A user always starts with a model conforming to the default metamodel shipped by DPF. This model is called an *enriched graph* in [136] and is written in terms of the following metamodel concepts – *DeepArrow*, *Inheritance*, *NodeProperty*, *Value*, *ArrowProperty*, *Containment*, *DeepNode*, *EString*, *EDataType* and *EBoolean*. From this point on models are always instantiated on classification levels further down.

User-defined notations for model elements are created using the so-called *visualization editor* and are then mapped to the concrete syntax. This approach follows the previously presented GMF approach which uses one model for concrete syntax modeling, one for abstract syntax modeling and one for mapping concrete and abstract syntax to each other. DPF's visualizer search algorithm is based on name matching and classification. Metamodel elements are assigned the *ids* specified for visualization descriptions in the visualization model. Based on this information, the mapping between abstract and concrete syntax is made. Visualization information is then mapped to instances via a wizard which takes the visualization model of the meta level as input and creates all information for instance visualization. The current implementation is somewhat limited, as evidenced by the limitations listed in [136] and does not attempt to support multi-notation, multi-format seamless editing.

12.5 Related Work Summary

As the previous sections demonstrate there are a lot of tool implementations and theoretical approaches that can be regarded as related work. The results of the previous discussions are summarized in Table 12.1 to give a condensed overview of the relationships between the analyzed tools and the deep, multi-format, multi-notation, seamless modeling approach presented in this thesis and realized by Melanee.

The summary shows that of the thirteen tools discussed in this chapter only Melanee fulfills all requirements outlined in the introduction. Only two of the tools are deep (MetaDepth and DPF) and therefore support seamless modeling. However, they do so but in an unassisted way, so this feature has been marked as partially supported by both tools in the table. Moreover, they do not support any of the other user-defined syntax features such as multi-format

		Deep	Seamless	Multi-format	Multi-notation	Aspect-oriented	Context Sensitive
Diagrammatic	GMF	✗	✗	✗	✗	✗	✓
	Sirius	✗	✗	(✓)	(✓)	✗	✓
	MetaEdit+	✗	✗	(✓)	(✓)	✗	✓
	AToMPM	✗	✗	✗	(✓)	✗	✓
	GME	✗	✗	✗	(✓)	✗	✗
Textual	XText	✗	✗	✗	✗	✗	✗
	MPS	✗	✗	✓	(✓)	✗	✗
	EMFText	✗	✗	✗	✗	✗	✗
Table	Parsley	✗	✗	✗	✗	✗	✗
	EMF Forms	✗	✗	✗	✗	✗	✗
	Sirius	✗	✗	(✓)	(✓)	✗	✗
Forms	Parsley	✗	✗	✗	✗	✗	✗
	EMF Forms	✗	✗	✗	✗	✗	✗
Deep	MetaDepth	✓	(✓)	✗	✗	✗	✗
	DPF	✓	(✓)	✗	✗	✗	✗
	Melanee	✓	✓	✓	✓	✓	✓

Table 12.1: Summary of the related work. Ticks in brackets indicate partial support.

and multi-notation modeling. Multi-format editing is partly supported by Sirius and MetaEdit+ which support diagrams and tables, but because this support is not available out-of-the-box for text the corresponding tick in Table 12.1 is placed in brackets. The only tool fully supporting multi-format editing is Jetbrains MPS. Multi-notation support is not present in any tool in such a way that within one model the notations can be exchanged and mixed as desired by a modeler. Sirius, MetaEdit+, AToMPM and GME support multi-notation in a limited form by allowing the whole model to be viewed in different notations. Aspect-oriented concrete syntax definition is not supported by any of the tools, while context sensitivity is supported only by the diagrammatic tools GMF, Sirius, MetaEdit+, and AToMPM. Deep constraint languages, per se, are not considered

in this related work chapter because they are an enabler for other features and not the focus of this work. Apart from Melanee, however, MetaDepth is the only tool providing a deep constraint language.

The related work overview shows that the most feature-rich tools are in the domain of diagrammatic, user-defined languages, with partial support for context-sensitive, multi-format, multi-notation user-defined languages. Nearly all tools from the other domains (text, table and forms) lack these features. As the table shows, the only tool which supports all features across all formats is Melanee.

Chapter 13

Conclusions & Future Work

This thesis has presented the deep, seamless, multi-format, multi-notation modeling approach for defining and using domain-specific languages implemented in the Melanee tool. The approach was developed to support the seven requirements identified in the introduction. Meeting these requirements allows the various stakeholders of a model to use the notation (R3) and format (R4) that best fits their needs. Moreover, the native support for deep domains (R1) and seamless modeling (R2) greatly simplifies the definition of languages by making changes to the abstract and concrete syntax immediately available to all classified levels. Deep, seamless modeling is backed up by a model evolution assistance service (i.e. emendation service) which helps the user effectively handle changes in models representing deep domains. The provided mechanism for user-defined visualization definition allows the visualization of models to depend on their current context/state (R5) and the definition of deep concrete syntax is driven by aspects (R6). Aspect-oriented visualization definition makes general, user-defined visualizations defined at abstract classification and inheritance levels easily refineable at more concrete levels. Finally, all user-defined visualization features are supported by a deep constraint language (R7).

13.1 Future Work

Although this work addresses most of the established requirements for a modern language workbench, some points are open for future work. The only way to define an execution semantics at the moment is to write Java code against the tool API or to write a deep transformation [25] to an executable model. More convenient ways of defining semantics are possible such as integrating the graphical fUML [183] approach (cf. XMOF [164] for EMF) or implementing a textual deep action language (as in MetaDepth or the ALF language for UML [208]). In an fUML like environment a modeler could graphically define the behavior of models across multiple classification levels using an abstract, programming-oriented language and a search algorithm similar to that used in the deep visualization mechanism developed in this thesis. This would enable the definition of general languages with general execution semantics and concrete syntax that can be further refined, using aspect-oriented technology, as needed for the scenario to be modeled. Execution would then naturally take place at lower classification levels without the need to apply workarounds such as stereotypes to type levels for visualizing execution. Moreover, the current execution state would not only be visualizable at the instance levels but the instance levels could also be used to pause and resume an execution.

In comparison to tools such as JetBrains MPS and XText, it is clear that the textual format supported by Melanee lacks features such as type system definition and interpreter definition. Adding these features to the approach, for example, would allow textual programming languages to be defined. Another interesting question is the extent to which other formats such as diagrams and tables can benefit from these features. In the current version of Melanee the textual format is supported by pure projectional editing. However, a hybrid parser/projection-based approach, as used in JetBrains MPS, can enhance the usability of the text editor.

Concerning the aspect-oriented, user-defined visualization approach it would be beneficial to support aspects for the attributes of metaclasses in language definitions. In a diagrammatic language, for example, this would allow the background color of a model element defined at O_0 to be overridden by a model element at O_1 without replacing the full visualization containing the

recolored background.

At the time of writing the potential of the approach is limited due to the fact that a language is always defined and used within one single deep model. To overcome this limitation it would be advantageous to be able to link distributed, deep models with one another. For example, a snippet of a user-defined language (e.g. the general company modeling language at O_0 of the running example) could be placed in a GIT repository on the internet and then linked by other models for reuse. To make the feature useable in practice the emendation service would have to be extended to support the emendation of remote models. This service could then transport emendation operations across cascades of linked models (i.e. linked models which link other models). Further research, however, has to be done on this topic.

When the modeling environment becomes distributed it is important to introduce rights management for deep models. In Melanee this could be achieved by extending application visualizers to control which groups of users can view and/or edit which parts of a model in certain notations and formats. Such a rights management approach would ensure that unauthorized modelers are unable to make changes to deep models which could negatively effect other linked models.

It would not only be interesting to extend the emendation service to support distributed models but also to support user-defined visualization definitions and their accompanying constraints. The current implementation of the user-defined visualization definition approach stores constraints as plain text but it would be possible to store pointers to deep model content wherever it is referenced in a constraint. A constraint would then be stored as a mix of static text and pointers to the abstract syntax model representation of a deep model. All other references to deep model content in a user-defined visualization definition are already stored as pointers in the current version. By having this information available, in future implementations the emendation service can calculate the impact of a change on user-defined syntax definitions and make suggestions for automatic emendation operations.

At the time of writing only the diagrammatic format persists its weaving model to store layout information. In all other formats user-defined layout customizations are lost after closing a format specific editor. A study on how

users actually work with the non-diagrammatic formats can give insights into the benefit of enriching these weaving models with layout information and persisting them for reuse once the format-specific editor is closed.

The current implementation of the approach is dominated by the diagrammatic editor. All other formats, for example, are invoked from the diagrammatic format. Hence, this format plays a more prominent role than others at the moment. In future it would be interesting to explore how this dominance of the diagrammatic format can be weakened and eventually removed altogether. One solution could be to display a dialog box prompting for the format and notation to be used when the deep model is opened. Another potential solution is to allow the application visualizers to be configured by the user for this purpose.

In Melanee, the tree format is also available for modeling but not further described here. This format is currently limited to the predefined tree modeling language offered by EMF out-of-the-box. In future versions of Melanee, this predefined language could be further tailored for deep modeling, e.g. by offering deep modeling features such as ontological instantiation through its user interface. It would also be possible to enrich this format with user-defined tree language definition capabilities.

Currently, Melanee is a desktop-based Eclipse application. However, with the advent of web-based Eclipse versions such as the Eclipse Remote Application Platform [67] or Eclipse Che [62] interesting future work would be to evaluate which parts of Melanee can be usefully ported to these web technologies.

13.2 Conclusions

At the time of writing the approach to user-defined language definition and use presented in this thesis is one of the most advanced available by addressing the in the introduction established requirements. The related work chapter shows that no tool currently exists which addresses all features offered by Melanee. The strict application of the visualizer search algorithm in a deep environment together with the strict application of the projectional editing approach across all modeling formats allows Melanee to offer a deep, seamless, multi-notation

and multi-format editing experience that is unlike anything currently available.

The evaluation shows that a whole stack of tools and technologies is needed to create a modeling workbench which is comparable to the approach developed as part of this thesis. A comparison of such tool stacks (e.g. the EMF tool stack) with Melanee shows that they introduce significant accidental complexity not only due to their reliance on workarounds to emulate deep modeling but also due to the heterogeneous technologies and tools that need to be integrated. In the textual format, for example, model merging problems between the parser-based textual editor and projectional editors of other formats would have to be solved to introduce new formats besides the textual one. The problems of a heterogeneous technology landscape are also highlighted by language evolution. For instance, different modeling language evolution tools have to be deployed to co-evolve the concrete syntax definitions in format-specific technologies and the abstract syntax definition in EMF. Moreover, although the EMF-based domain model can represent the content in the running example, it lacks deep features such as control of attribute mutability etc.

In conclusion, we believe the deep user-defined language technology developed in this thesis and implemented in Melanee represents a significant contribution to the state of the art, and hope it will form the basis for more useful, flexible and exciting modeling features in the future.

Bibliography

[1] Paul W. Abrahams. A final solution to the dangling else of algol 60 and related languages. *Commun. ACM*, 9(9):679–682, September 1966. pages x, 67

[2] Acceleo. Project Website. `http://www.eclipse.org/acceleo`, 2016. pages 243

[3] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972. pages 66

[4] Timo Asikainen and Tomi Männistö. Nivel: A metamodelling language with a formal semantics. *Software & Systems Modeling*, 8(4):521–549, 2009. pages 245, 246

[5] Colin Atkinson. Meta-modelling for distributed object environments. In *Enterprise Distributed Object Computing Workshop [1997]. EDOC '97. Proceedings. First International*, pages 90–101, 1997. pages 20

[6] Colin Atkinson. Supporting and applying the uml conceptual framework. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language. Â«UMLÂ»'98: Beyond the Notation: First International Workshop, Mulhouse, France, June 3-4, 1998. Selected Papers*, pages 21–36, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. pages 18

[7] Colin Atkinson, Florian Barth, Ralph Gerbig, Felix Freiling, Sebastian Schinzel, Frank Hadasch, Alexander Maedche, and Benjamin Muller. Reducing the incidence of unintended, human-caused information flows in enterprise systems. In *Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2012 IEEE 16th International*, pages 11–18, 2012. pages 9, 41

[8] Colin Atkinson and Ralph Gerbig. Melanie: Multi-level modeling and ontology engineering environment. In *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards*,

MW '12, pages 7:1–7:2, New York, NY, USA, 2012. ACM. pages 3, 9, 27, 161

[9] Colin Atkinson and Ralph Gerbig. Harmonizing textual and graphical visualizations of domain specific models. In *Proceedings of the Second Workshop on Graphical Modeling Language Development*, GMLD '13, pages 32–41, New York, NY, USA, 2013. ACM. pages 9

[10] Colin Atkinson and Ralph Gerbig. Level-agnostic designation of model elements. In Jordi Cabot and Julia Rubin, editors, *Modelling Foundations and Applications*, volume 8569 of *Lecture Notes in Computer Science*, pages 18–34. Springer International Publishing, 2014. pages 9, 52

[11] Colin Atkinson and Ralph Gerbig. Aspect-oriented concrete syntax definition for deep modeling languages. In Colin Atkinson, Georg Grossmann, Thomas Kühne, and Juan de Lara, editors, *Proceedings of the 2nd International Workshop on Multi-Level Modelling co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2015)*, volume CEUR-WS.org/Vol-1505 of *Multi 2015*, pages 13–22, 2015. pages 9, 42, 44

[12] Colin Atkinson and Ralph Gerbig. A feature-based comparison of melanee and metadepth. In Colin Atkinson and, editor, *Proceedings of the 3rd International Workshop on Multi-Level Modelling co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2016)*, Multi 2016, 2016. pages 9, 72, 245

[13] Colin Atkinson and Ralph Gerbig. Flexible deep modeling with melanee. In *Modellierung 2016 (Workshops)*, volume 255, pages 117–122, 2016. pages 9

[14] Colin Atkinson, Ralph Gerbig, and Mathias Fritzsche. Modeling language extension in the enterprise systems domain. In *Enterprise Distributed Object Computing Conference (EDOC), 2013 17th IEEE International*, pages 49–58, 2013. pages 3, 9

[15] Colin Atkinson, Ralph Gerbig, and Mathias Fritzsche. A multi-level approach to modeling language extension in the enterprise systems domain. *Information Systems*, 2015. pages 3, 9, 38, 41, 67

[16] Colin Atkinson, Ralph Gerbig, and Bastian Kennel. On-the-fly emendation of multi-level models. In *Proceedings of the 8th European Conference on Modelling Foundations and Applications*, ECMFA'12, pages 194–209, Berlin, Heidelberg, 2012. Springer-Verlag. pages xi, 4, 9, 142, 143, 148, 158

[17] Colin Atkinson, Ralph Gerbig, and Bastian Kennel. Symbiotic general-purpose and domain-specific languages. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1269–1272, June 2012. pages 9, 27, 37, 213

[18] Colin Atkinson, Ralph Gerbig, and Thomas Kühne. Comparing multi-level modeling approaches. In Colin Atkinson, Georg Grossmann, Thomas Kühne, and Juan de Lara, editors, *Proceedings of the Workshop on Multi-Level Modelling co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014)*, volume CEUR-WS.org/Vol-1286, pages 53–61, 2014. pages 9

[19] Colin Atkinson, Ralph Gerbig, and Thomas Kühne. Opportunities and challenges for deep constraint languages. In Achim D. Brucker, Marina Egea, Martin Gogolla, and Frédéric Tuong, editors, *OCL 2015 – 15th International Workshop on OCL and Textual Modeling: Tools and Textual Model Transformations Workshop Proceedings*, volume 1512 of *OCL 2015*, pages 3–18, 2015. pages 9, 115, 121

[20] Colin Atkinson, Ralph Gerbig, and Thomas Kühne. A unifying approach to connections for multi-level modeling. In *Proceedings of the 18th international conference on Model driven engineering languages and systems*, MODELS 2015, 2015. pages 9, 54, 118, 146

[21] Colin Atkinson, Ralph Gerbig, Katharina Markert, Mariia Zrianina, Alexander Egurnov, and Fabian Kajzar. Towards a deep, domain specific modeling framework for robot applications. In *Proceedings of the*

1st International Workshop on Model-Driven Robot Software Engineering co-located with International Conference on Software Technologies: Applications and Foundations (STAF 2014), MORSE '14, pages 4–15. CEUR-WS.org, 2014. pages 9, 41, 44, 163

[22] Colin Atkinson, Ralph Gerbig, and Noah Metzger. On the execution of deep models. In *Proceedings of the 1st International Workshop on Executable Modeling co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015)*, volume 1560, pages 28–33. CEUR Workshop Proceedings, 2015. pages 9, 48, 163

[23] Colin Atkinson, Ralph Gerbig, and Christian Tunjic. Towards multi-level aware model transformations. In Zhenjiang Hu and Juan de Lara, editors, *Theory and Practice of Model Transformations*, volume 7307 of *Lecture Notes in Computer Science*, pages 208–223. Springer Berlin Heidelberg, 2012. pages 9

[24] Colin Atkinson, Ralph Gerbig, and Christian Tunjic. A multi-level modeling environment for sum-based software engineering. In *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, VAO '13, pages 2:1–2:9, New York, NY, USA, 2013. ACM. pages 9, 163

[25] Colin Atkinson, Ralph Gerbig, and ChristianVjekoslav Tunjic. Enhancing classic transformation languages to support multi-level modeling. *Software & Systems Modeling*, pages 1–22, 2013. pages 9, 121, 252

[26] Colin Atkinson, Matthias Gutheil, and Bastian Kennel. A flexible infrastructure for multilevel language engineering. *Software Engineering, IEEE Transactions on*, 35(6), 2009. pages 26, 161

[27] Colin Atkinson, Bastian Kennel, and Björn Goß. The level-agnostic modeling language. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 266–275. Springer Berlin Heidelberg, 2011. pages 19, 26, 52

[28] Colin Atkinson, Bastian Kennel, and Björn Goß. Supporting constructive and exploratory modes of modeling in multi-level ontologies. In *Proceedings 7th Int. Workshop on Semantic Web Enabled Software Engineering*, SWESE, pages 1–15, 2011. pages 158

[29] Colin Atkinson and Thomas Kühne. The essence of multilevel metamodeling. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools: 4th International Conference Toronto, Canada, October 1–5, 2001 Proceedings*, pages 19–33, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. pages 20

[30] Colin Atkinson and Thomas Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41, September 2003. pages 9, 18

[31] Colin Atkinson and Thomas Kühne. Reducing accidental complexity in domain models. *Software & Systems Modeling*, 7(3):345–359, 2008. pages 2, 191

[32] John W Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. *Proceedings of the International Comference on Information Processing, 1959*, 1959. pages 14

[33] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Eng.*, 28(1):4–17, January 2002. pages 194, 196, 199

[34] Michael Barr and Charles Wells. *Category theory for computing science*, volume 49. Prentice Hall New York, 1990. pages 246

[35] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. First Experiments with a ModelWeaver. In *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004. pages 33

[36] Xavier Blanc, Isabelle Mounier, Alix Mougenot, and Tom Mens. Detecting model inconsistency through operation-based model construction. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 511–520, 2008. pages 148, 149

[37] Claus Brabrand, Robert Giegerich, and Anders MÃ¸ller. Analyzing ambiguity of context-free grammars. *Science of Computer Programming*, 75(3):176 – 191, 2010. pages 67

[38] Frederick P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987. pages 2, 44, 191

[39] Petra Brosch, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Philip Langer. The operation recorder: specifying model refactorings by-example. In *OOPSLA Companion*, pages 791–792, 2009. pages 4, 143

[40] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM. pages 65

[41] Fabien Campagne. *The MPS Language Workbench Volume I: The Meta Programming System*. Fabien Campagne, first edition, 2014. pages 4, 240, 242

[42] David G. Cantor. On the ambiguity problem of backus systems. *J. ACM*, 9(4):477–479, October 1962. pages 67

[43] Cederic Brun. EcoreTools 2.0: The Making-Of. `http://www.youtube.com/watch?v=XSP-oAmmS_E`, 2013. pages 235

[44] Kai Chen, Janos Sztipanovits, and Sandeep Neema. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, pages 35–43, New York, NY, USA, 2005. ACM. pages 14

[45] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976. pages 5, 52

[46] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994. pages 194

[47] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating co-evolution in model-driven engineering. In *Enterprise Distributed Object Computing Conference, 2008. EDOC '08. 12th International IEEE*, pages 222–231, 2008. pages 148

[48] Tony Clark, Paul Sammut, and James S. Willans. Applied metamodelling: A foundation for language driven development (third edition). *CoRR*, 2015. pages 245

[49] Tony Clark and James Willans. Software language engineering with xmf and xmodeler. *Formal and Practical Aspects of Domain Specific Languages: Recent Developments*, 2013. pages 234, 245

[50] Juan de Lara and Esther Guerra. Deep meta-modelling with metadepth. In *Proceedings of the 48th international conference on Objects, models, components, patterns*, TOOLS'10, pages 1–20, Berlin, Heidelberg, 2010. Springer-Verlag. pages 3, 18, 71, 72, 245

[51] Juan de Lara and Esther Guerra. Domain-specific textual metamodelling languages for model driven engineering. In Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitris Kolovos, editors, *Modelling Foundations and Applications*, volume 7349 of *Lecture Notes in Computer Science*, pages 259–274. Springer Berlin / Heidelberg, 2012. pages 2, 245

[52] Juan de Lara and Esther Guerra. Towards automating the analysis of integrity constraints in multi-level models. In *Proceedings of the 1st International Workshop on Multi-Level Modelling co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014)*, volume 1286, pages 63–72, 2014. pages 246

[53] Juan de Lara, Esther Guerra, Ruth Cobos, and Jaime Moreno-Llorena. Extending deep meta-modelling for practical model-driven engineering. *The Computer Journal*, 2012. pages 41

[54] Juan de Lara, Esther Guerra, Ruth Cobos, and Jaime Moreno-Llorena. Extending deep meta-modelling for practical model-driven engineering. *The Computer Journal*, 57(1):36–58, 2014. pages 245

[55] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. Model-driven engineering with domain-specific meta-modelling languages. *Software & Systems Modeling*, 14(1):429–459, 2015. pages 41, 42

[56] Juan de Lara, Esther Guerra, and Sánchez Cuarado Jesús. A-posteriori typing for model-driven engineering. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, MODELS 2015, pages 156–165, 2015. pages 143, 158, 159

[57] Andreas Demuth, Roberto E. Lopez-Herrejon, and Alexander Egyed. Cross-layer modeler: A tool for flexible multilevel modeling with consistency checking. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 452–455, New York, NY, USA, 2011. ACM. pages 143, 159, 245

[58] Davide Di Ruscio, Ralf Lämmel, and Alfonso Pierantonio. Automated co-evolution of gmf editor models. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering: Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*, pages 143–162. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. pages 159, 207

[59] Zinovy Diskin and Boris Kadish. Variable set semantics for keyed generalized sketches: Formal semantics for object identity and abstract syntax for conceptual modeling. *Data & Knowledge Engineering*, 47(1):1–59, October 2003. pages 246

[60] DSM Forum. Dsm tools. `http://www.dsmforum.org/tools.html`, 2015. pages 1

[61] Marlon Dumas and ArthurH.M. ter Hofstede. Uml activity diagrams as a workflow specification language. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 76–90. Springer Berlin Heidelberg, 2001. pages 23

[62] Eclipse Che. Project Website. `http://www.eclipse.org/che/`, 2016. pages 254

[63] Eclipse Foundation. Sirius with Cedric Brun, Obeo - EclipseCon France. `https://www.youtube.com/watch?v=hyDxSmbSi2g`, 2013. pages 235

[64] Eclipse Foundation. Xtext documentation - integration with gmf editors. `http://www.eclipse.org/Xtext/documentation/308_emf_integration.html#gmf-integration`, 2016. pages 216

[65] Eclipse Foundation. Project diversity. `http://dash.eclipse.org/dash/commits/web-app/project-diversity.cgi`, December 2012. pages 1

[66] Eclipse Modeling Project. Project Website. `https://eclipse.org/modeling/`, 2016. pages 178

[67] Eclipse Remote Application Platform. Project Website. `http://www.eclipse.org/rap/`, 2016. pages 254

[68] Edapt. Project Website. `http://www.eclipse.org/edapt/`, 2015. pages 205

[69] Sven Efftinge, Peter Friese, Arno Hase, Dennis Hübner, Clemens Kadura, Bernd Kolb, Jan Köhnlein, Dieter Moroff, Karsten Thoms, Markus Völter, et al. Xpand documentation. `http://git.eclipse.org/c/m2t/org.eclipse.xpand.git/plain/doc/org.eclipse.xpand.doc/manual/xpand_reference.pdf`, 2014. pages 162, 235

[70] Sven Efftinge and Markus Völter. oaw xtext: A framework for textual dsls. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, page 118, 2006. pages 16, 215, 240, 241

[71] Karsten Ehrig, Claudia Ermel, Stefan Hänsgen, and Gabriele Taentzer. Generation of visual editors as eclipse plug-ins. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 134–143, New York, NY, USA, 2005. ACM. pages 54, 234

[72] EMF Forms. Project website. `https://www.eclipse.org/ecp/emfforms/`, 2016. pages 16, 244

[73] EMFText. EMFText User Guide. `http://www.emftext.org/EMFTextGuide.php`, 2012. pages 243

[74] EMFText. Project Website. `http://www.emftext.org`, 2016. pages 243

[75] Rik Eshuis. Reconciling statechart semantics. *Science of Computer Programming*, 74(3):65 – 99, 2009. pages 16

[76] Javier Espinazo-Pagán, Marcos Menárguez, and Jesús García-Molina. Metamodel syntactic sheets: An approach for defining textual concrete syntaxes. In *Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA '08, pages 185–199, Berlin, Heidelberg, 2008. Springer-Verlag. pages 26, 241, 246

[77] Eugene Syriani. AToMPM - Models 2013 Demonstrations. `http://www.youtube.com/watch?v=iBbdpmpwn6M`, 2013. pages 238

[78] Moritz Eysholdt. Converging Textual and Graphical Editors. In *Eclipse Modeling Days 2009*. itemis, 2009. pages 68

[79] Jean-Marie Favre. Languages evolve too! changing the software time scale. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, IWPSE '05, pages 33–44, Washington, DC, USA, 2005. IEEE Computer Society. pages 3

[80] Martin Fowler. Projectional editing. `http://martinfowler.com/bliki/ProjectionalEditing.html`, 2008. pages 66

[81] DPF: Diagram Predicate Framework. Project Website. `http://dpf.hib.no`, 2016. pages 246

[82] Ulrich Frank. Multi-perspective enterprise modeling: foundational concepts, prospects and future research challenges. *Software & Systems Modeling*, 13(3):941–962, 2014. pages 2

[83] Ulrich Frank. Multilevel modeling. *Business & Information Systems Engineering*, 6(6):319–337, 2014. pages 245

[84] Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. Managing model adaptation by precise detection of metamodel changes. In RichardF. Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 34–49. Springer Berlin Heidelberg, 2009. pages 148

[85] Jokin García, Oscar Diaz, and Maider Azanza. Model transformation co-evolution: A semi-automatic approach. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering*, volume 7745 of *Lecture Notes in Computer Science*, pages 144–163. Springer Berlin Heidelberg, 2013. pages 159

[86] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A semantic framework for metamodel-based languages. *Automated Software Engineering*, 16(3-4):415–454, 2009. pages 14

[87] Gartner. Gartner's 2006 emerging technologies hype cycle highlights key technology themes. `http://www.gartner.com/it/page.jsp?id=495475`, 2006. pages 1

[88] gentleware. Poseidon for DSLs 2.0. `http://www.gentleware.com/poseidon-for-dsls.html`, 2014. pages 234

[89] Ralph Gerbig. The level-agnostic modeling language: Language specification and tool implementation. Master's thesis, University of Mannheim, Mannheim, Germany, 2011. pages 3, 22, 26, 27, 52

[90] GME: Generic Modeling Environment. Project Website. `http://www.isis.vanderbilt.edu/projects/gme/`, 2016. pages 239

[91] Cesar Gonzalez-Perez. Tools for an extended object modelling environment. In *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, pages 20–23, June 2005. pages 245

[92] Cesar Gonzalez-Perez and Brian Henderson-Sellers. A powertype-based metamodelling framework. *Software & Systems Modeling*, 5(1):72–90, 2006. pages 2, 41

[93] Saul Gorn. Specification languages for mechanical languages and their processors a baker's dozen: A set of examples presented to asa x3.4 subcommittee. *Commun. ACM*, 4(12):532–542, 1961. pages 14

[94] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 1 edition, 2009. pages ix, 16, 33, 54, 60, 61, 162, 234

[95] Boris Gruschko, Dimitrios Kolovos, and Richard Paige. Towards synchronizing models with evolving metamodels. In *Proceedings of the International Workshop on Model-Driven Software Evolution*, 2010. pages 147

[96] Esther Guerra and Juan de Lara. Automated analysis of integrity constraints in multi-level models. *Data & Knowledge Engineering*, 2016. pages 159, 246

[97] Matthias Gutheil, Bastian Kennel, and Colin Atkinson. A systematic approach to connectors in a multi-level modeling environment. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 843–857. Springer Berlin Heidelberg, 2008. pages 54, 146

[98] Wilfred James Hansen. *Creation of hierarchic text with a computer display*. PhD thesis, Stanford University, Stanford, CA, USA, 1971. pages 66

[99] Wilfred James Hansen. User engineering principles for interactive systems. In *Proceedings of the November 16-18, 1971, Fall Joint Computer*

Conference, AFIPS '71 (Fall), pages 523–532, New York, NY, USA, 1971. ACM. pages 66

[100] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987. pages 16

[101] David Harel and Bernhard Rumpe. Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37(10):64–72, 2004. pages 14

[102] Xiao He, Zhiyi Ma, Weizhong Shao, and Ge Li. A metamodel for the notation of graphical modeling languages. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 1, pages 219–224, 2007. pages 54

[103] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and refinement of textual syntax for models. In RichardF. Paige, Alan Hartman, and Arend Rensink, editors, *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 114–129. Springer Berlin Heidelberg, 2009. pages 16, 240, 243

[104] Brian Henderson-Sellers, Tony Clark, and Cesar Gonzalez-Perez. On the search for a level-agnostic modelling language. In Camille Salinesi, MoiraC. Norrie, and Oscar Pastor, editors, *Advanced Information Systems Engineering*, volume 7908 of *Lecture Notes in Computer Science*, pages 240–255. Springer Berlin Heidelberg, 2013. pages 245

[105] Brian Henderson-Sellers and Cesar Gonzalez-Perez. The rationale of powertype-based metamodelling to underpin software development methodologies. In *Proceedings of the 2Nd Asia-Pacific Conference on Conceptual Modelling - Volume 43*, APCCM '05, pages 7–16, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc. pages 18

[106] Brian Henderson-Sellers and Bhuvan Unhelkar. *Open Modeling with UML*. Addison-Wesley, 2000. pages ix, 17

[107] Markus Herrmannsdoerfer. Cope – a workbench for the coupled evolution of metamodels and models. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 286–295. Springer Berlin Heidelberg, 2011. pages 143

[108] Markus Herrmannsdoerfer. Solving the TTC 2011 model migration case with edapt. In *Proceedings Fifth Transformation Tool Contest, TTC 2011, Zürich, Switzerland, June 29-30 2011.*, pages 27–35, 2011. pages 4, 143, 205

[109] Markus Herrmannsdoerfer, Daniel Ratiu, and Guido Wachsmuth. Language evolution in practice: The history of gmf. In Mark van den Brand, Dragan Gašević, and Jeff Gray, editors, *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 3–22. Springer Berlin Heidelberg, 2010. pages 3, 4

[110] Markus Herrmannsdoerfer, SanderD. Vermolen, and Guido Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 163–182. Springer Berlin Heidelberg, 2011. pages 147, 149

[111] Shuhei Hiya, Kenji Hisazumi, Akira Fukuda, and Tsuneo Nakanishi. clooca: Web based tool for domain specific modeling. In *Demos/Posters/StudentResearch@ MoDELS*, pages 31–35, 2013. pages 234

[112] Eero Hyvönen and Eetu Mäkelä. Semantic autocompletion. In Riichiro Mizoguchi, Zhongzhi Shi, and Fausto Giunchiglia, editors, *The Semantic Web – ASWC 2006: First Asian Semantic Web Conference, Beijing, China, September 3-7, 2006. Proceedings*, pages 739–751, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. pages 65

[113] Muzaffar Igamberdiev, Georg Grossmann, Matt Selway, and Markus Stumptner. An integrated multi-level modeling approach for industrial-scale data interoperability. *Software & Systems Modeling*, pages 1–26, 2016. pages 41

[114] Keith Instone. Location, path and attribute breadcrumbs. *3rd Annual Information Architecture Summit*, pages 15–17, 2002. pages 88

[115] Intentional. Technology. `http://www.intentsoft.com/intentional-technology/`, 2015. pages 4, 71, 234

[116] Pontus Johnson, Johan Ullberg, Markus Buschle, Ulrik Franke, and Khurram Shahzad. P2amf: Predictive, probabilistic architecture modeling framework. In Marten van Sinderen, Paul Oude Luttighuis, Erwin Folmer, and Steven Bosems, editors, *Enterprise Interoperability: 5th International IFIP Working Conference, IWEI 2013, Enschede, The Netherlands, March 27-28, 2013. Proceedings*, pages 104–117, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. pages 125

[117] Stephen C Johnson. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975. pages 14

[118] Stephen C Johnson. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975. pages 67

[119] Andreas Jordan, Georg Grossmann, Wolfgang Mayer, Matt Selway, and Markus Stumptner. On the application of software modelling principles on iso 15926. In *Proceedings of the Modelling of the Physical World Workshop*, MOTPW '12, pages 3:1–3:6, New York, NY, USA, 2012. ACM. pages 2

[120] Frédéric Jouault and Jean Bézivin. On the specification of textual syntaxes for models. In *Eclipse Modeling Symposium at the first Eclipse Summit Europe*, 2006. pages 241

[121] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. Tcs: A dsl for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, pages 249–254, New York, NY, USA, 2006. ACM. pages 241

[122] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Confer-*

ence, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer Berlin Heidelberg, 2006. pages 121, 241

[123] JRebel Labs. Java tools & technologies landscape for 2014. `http://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/6/`, 2013. pages 1

[124] Dominik Kantner. *Specification and implementation of a deep ocl dialect.* PhD thesis, Fakultät für Wirtschaftsinformatik und Wirtschaftsmathematik, 2014. pages 114, 115, 121, 224

[125] Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: Rules for declarative specification of languages and ides. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463, New York, NY, USA, 2010. ACM. pages 14, 241

[126] Steven Kelly. Modeling tools history. `https://twitter.com/stevekmcc/status/454174783966429184/photo/1`, 2014. pages 1

[127] Bastian Kennel. *A Unified Framework for Multi-level Modeling.* PhD thesis, University of Mannheim, 2012. pages 9, 22, 52, 54, 142, 143, 145, 162, 178

[128] Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, Jacques Robin, and Marie-Pierre Gervais. Detecting complex changes during metamodel evolution. In Jelena Zdravkovic, Marite Kirikova, and Paul Johannesson, editors, *Advanced Information Systems Engineering*, volume 9097 of *Lecture Notes in Computer Science*, pages 263–278. Springer International Publishing, 2015. pages 3, 4, 147, 148, 149, 157, 158

[129] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming: 15th European Conference Budapest, Hungary, June 18–22, 2001 Proceedings*, pages 327–354, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. pages 42

[130] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings*, pages 220–242, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. pages 42

[131] Lars Kristian Klauske and Christian Dziobek. Improving modeling usability: Automated layout generation for simulink. In *Proceedings of the MathWorks Automotive Conference, MAC*, 2010. pages 51

[132] Michel Klein and Natalya F. Noy. A component-based framework for ontology evolution. In *Workshop on Ontologies and Distributed Systems at IJCAI-03*, 2003. pages 4, 143, 158

[133] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 1 edition, 2008. pages 13, 14, 16

[134] Paul Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(2):176–201, 1993. pages 14

[135] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(3):331–380, July 2005. pages 14

[136] Ole Klokkhammer. A diagrammatic approach to deep metamodelling. Master's thesis, University of Bergen, 2014. pages 246, 247

[137] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The epsilon object language (eol). In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture – Foundations and Applications: Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006. Proceedings*, pages 128–142. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. pages 245

[138] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The epsilon transformation language. In Antonio Vallecillo, Jeff Gray, and

Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations: First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008 Proceedings*, pages 46–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. pages 245

[139] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. On the evolution of ocl for capturing structural constraints in modelling languages. In Jean-Raymond Abrial and Uwe Glässer, editors, *Rigorous Methods for Software Construction and Analysis: Essays Dedicated to Egon Börger on the Occasion of His 60th Birthday*, pages 204–218. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. pages 6, 162, 245

[140] Dimitrios S. Kolovos, Louis M. Rose, Saad Bin Abid, Richard F. Paige, Fiona A. C. Polack, and Goetz Botterweck. Taming emf and gmf using model transformation. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems: 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, pages 211–225. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. pages 234

[141] Dimitris Kolovos, Nicholas Matragkas, and Antonio Garcia-Dominguez. Towards flexible parsing of structured textual model representations. *2nd Flexible Model Driven Engineering Proceedings (FlexMDE 2016)*, 2016. pages 205

[142] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372, 2010. pages 241

[143] Andrew Krause. *Foundations of GTK+ development*. Apress, 2007. pages 101

[144] Jarosław Krochmalski. *IntelliJ IDEA Essentials*. Packt Publishing Ltd, 2014. pages 241

[145] Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. Extensive validation of ocl models by integrating sat solving into use. In Judith

Bishop and Antonio Vallecillo, editors, *Objects, Models, Components, Patterns: 49th International Conference, TOOLS 2011, Zurich, Switzerland, June 28-30, 2011. Proceedings*, pages 290–306, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. pages 246

[146] 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. pages 18, 245

[147] Thomas Kühne and Friedrich Steimann. Tiefe charakterisierung. In *Modellierung 2004, Proceedings zur Tagung, 23.-26. März 2004, Marburg, Proceedings*, pages 109–119, 2004. pages 18

[148] Yngve Lamo, Xiaoliang Wang, Florian Mantz, Oyvind Bech, Anders Sandven, and Adrian Rutle. Dpf workbench: A multi-level language workbench for mde. *Proceedings of the Estonian Academy of Sciences, 2013*, 62(1):3–15, 2013. pages 3, 246

[149] Bernard Lang. On the usefulness of syntax directed editors. In Reidar Conradi, Tor M. Didriksen, and Dag H. Wanvik, editors, *Advanced Programming Environments: Proceedings of an International Workshop Trondheim, Norway, June 16–18, 1986*, pages 47–51. Springer Berlin Heidelberg, Berlin, Heidelberg, 1986. pages 66, 71

[150] Philip Langer, Manuel Wimmer, Petra Brosch, Markus Herrmannsdörfer, Martina Seidl, Konrad Wieland, and Gerti Kappel. A posteriori operation detection in evolving software models. *The Journal of Systems and Software*, 86(2):551–566, 2013. pages 148

[151] Juan De Lara, Esther Guerra, and Jesús Sánchez Cuadrado. When and how to use multilevel modelling. *ACM Trans. Softw. Eng. Methodol.*, 24(2):12:1–12:46, December 2014. pages 2, 182, 198, 231

[152] Juan de Lara and Hans Vangheluwe. Atom3: A tool for multi-formalism and meta-modelling. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306

of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin Heidelberg, 2002. pages 238

[153] Jill H. Larkin and Herbert A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1):65–100, 1987. pages 51, 65

[154] Akos Ledeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, Nov 2001. pages 239

[155] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *Proceedings of WISP'2001*, WISP'2001. IEEE, 2001. pages 239

[156] D. Leroux, M. Nally, and K. Hussey. Rational software architect: A tool for domain-specific modeling. *IBM Systems Journal*, 45(3):555–568, 2006. pages 143

[157] Greg Little and Robert C. Miller. Keyword programming in java. *Automated Software Engineering*, 16(1):37–71, 2008. pages 65

[158] Mark Lorenz and Jeff Kidd. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., 1994. pages 194

[159] Tom F. Lunney and Ron H. Perrott. Syntax-directed editing. *Software Engineering Journal*, 3(2):37–46, 1988. pages 66

[160] Haohai Ma, Weizhong Shao, Lu Zhang, Zhiyi Ma, and Yanbing Jiang. Applying oo metrics to assess uml meta-models. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *Â«UMLÂ» 2004 — The Unified Modeling Language. Modeling Languages and Applications: 7th International Conference, Lisbon, Portugal, October 11-15, 2004. Proceedings*, pages 12–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. pages 194, 196

[161] Fernando Macias, Adrian Rutle, and Volker Stolz. Multecore: Combining the best of fixed-level and multilevel metamodelling. In Colin Atkinson, Georg Grossmann, and Tony Clark, editors, *Proceedings of the 3rd International Workshop on Multi-Level Modelling co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2016)*, volume CEUR-WS.org/Vol-1722 of *Multi 2016*, pages 65–75, 2016. pages 245

[162] F. A. Mann. Ieee standard logic symbols, why use them? In *Compcon Spring '88. Thirty-Third IEEE Computer Society International Conference, Digest of Papers*, pages 338–341, 1988. pages 36

[163] Raphael Manna. *A Multi-Paradigm Modelling Approach to theFoundations of Domain-Specic Modelling*. PhD thesis, McGill University, 2012. pages 238

[164] Tanja Mayerhofer, Philip Langer, Manuel Wimmer, and Gerti Kappel. xmof: Executable dsmls based on fuml. In Martin Erwig, RichardF. Paige, and Eric Van Wyk, editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 56–75. Springer International Publishing, 2013. pages 5, 252

[165] Diana Maynard, Wim Peters, Marta Sabou, and Mathieu dÁquin. Change management for metadata evolution. In *International Workshop on Ontology Dynamics (IWOD) ESWC 2007 Workshop*, 2007. pages 4, 143

[166] Metacase. Defining modeling languages and code generators in MetaEdit+. `http://www.youtube.com/watch?v=tCJIZtoeIPQ`, 2012. pages 237

[167] MetaDepth. Project Website. `http://www.metadepth.org`, 2016. pages 71, 245

[168] G. Mezei. *Transformation-based Support for Visual Languages*. Lambert Academic Publishing, first edition, 2010. pages 234

[169] Microsoft Developer Network. Modeling SDK for Visual Studio - Domain-Specific Languages. https://msdn.microsoft.com/en-us/library/bb126259.aspx, 2016. pages 234

[170] Daniel L. Moody. The "physics"; of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779, 2009. pages 51, 55

[171] MWE2. Project Website. https://eclipse.org/Xtext/documentation/306_mwe2.html, 2016. pages 242

[172] Steve Northover and Mike Wilson. *Swt: The Standard Widget Toolkit, Volume 1.* Addison-Wesley Professional, first edition, 2004. pages 101

[173] Natalya F. Noy and Michel Klein. Ontology evolution: Not the same as schema evolution. *Knowledge and Information Systems*, 6(4):428–440, 2004. pages 143, 149

[174] Object Management Group. Business Process Model and Notation (BPMN) Version 2.0. http://www.omg.org/spec/BPMN/2.0, 2011. pages 23

[175] Object Management Group. Diagram Definition (DD), Version 1.0. http://www.omg.org/spec/DD/1.0/, 2012. pages 54

[176] Object Management Group. MDA Guide, Version 1.0.1. http://www.omg.org/news/meetings/workshops/UML_2003_Manual/00-2_MDA_Guide_v1.0.1.pdf, 2013. pages 13

[177] OCLinEcore. Project Website. https://wiki.eclipse.org/OCL/OCLinEcore, 2013. pages 162

[178] OMG. Human-usable textual notation (hutn) specification. http://www.omg.org/spec/UML/2.4.1, 2004. pages 71, 245

[179] OMG. Diagram Interchange, Version 1.0.0. http://www.omg.org/cgi-bin/doc?formal/06-04-04, 2006. pages 33

[180] OMG. Mof model to text transformationÂ language^{TM} (mofm2t^{TM}), 1.0. http://www.omg.org/spec/MOFM2T/1.0/, 2008. pages 243

[181] OMG. OMG Unified Modeling Language ™ (OMG UML), Superstructure Version 2.4.1. http://www.omg.org/spec/UML/2.4.1, 2011. pages 16

[182] OMG. OMG Unified Modeling Language ™ (OMG UML), Version 2.5. http://www.omg.org/spec/UML/2.5, 2011. pages 52, 54

[183] OMG. Semantics of a foundational subset for executable uml models (fuml). http://www.omg.org/spec/FUML/1.1/, 2013. pages 252

[184] OMG. Object constraint language version 2.4. http://www.omg.org/spec/OCL/2.4/PDF, 2014. pages 6, 162, 224, 228

[185] OMG. Meta object facility (mof) 2.0 query/view/transformation specification. http://www.omg.org/spec/QVT/1.3/, 2016. pages 235

[186] William F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992. pages 149

[187] Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos, and Fion A. C. Polack. The design of a conceptual framework and technical infrastructure for model management language engineering. In *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*, pages 162–171, 2009. pages 162, 245

[188] Papyrus for Real Time (Papyrus-RT). Project Website. https://projects.eclipse.org/projects/modeling.papyrus-rt, 2016. pages 4

[189] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013. pages 67, 216

[190] Terence Parr and Kathleen Fisher. Ll(*): The foundation of the antlr parser generator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 425–436, New York, NY, USA, 2011. ACM. pages 67

[191] EMF Parsley. Project Website. http://www.eclipse.org/emf-parsley/, 2016. pages 16, 216, 244

[192] Marian Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Commun. ACM*, 38(6):33–44, June 1995. pages 51, 65

[193] EMF Client Platform. Project Website. `http://www.eclipse.org/ecp/`, 2016. pages 244

[194] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In Takayasu Ito and AlbertR. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 244–264. Springer Berlin Heidelberg, 1991. pages 16

[195] Maxime Porhel. Sirius + xtext = ♥. In *EclipseCon 2015*, 2015. pages 235

[196] Ernesto Posse. Papyrusrt: Modelling and code generation. In *Proceedings of the International Workshop on Open Source Software for Model Driven Engineering co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015)*, volume 1541, pages 54–63. CEUR Workshop Proceedings, 2015. pages 4

[197] Aurelien Pupier. Customize your gmf editor by customizing templates. `http://community.bonitasoft.com/customize-your-gmf-editor-customizing-templates`, 2010. pages 162, 235

[198] Sandeep Purao and Vijay Vaishnavi. Product metrics for object-oriented systems. *ACM Comput. Surv.*, 35(2):191–221, 2003. pages 194

[199] Jan Reimann, Mirko Seifert, and Uwe Aßmann. Role-based generic model refactoring. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part II*, MODELS 2010, pages 78–92, Berlin, Heidelberg, 2010. Springer-Verlag. pages 4, 143

[200] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Model migration with epsilon flock. In *Proceedings of the Third*

International Conference on Theory and Practice of Model Transformations, ICMT'10, pages 184–198, Berlin, Heidelberg, 2010. Springer-Verlag. pages 143

[201] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. The epsilon generation language. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture – Foundations and Applications: 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings*, pages 1–16. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. pages 245

[202] Alessandro Rossini. *Diagram Predicate Framework meets Model Versioning and Deep Metamodelling*. PhD thesis, University of Bergen, 2010. pages 246

[203] Alessandro Rossini, Juan de Lara, Esther Guerra, and Nikolay Nikolov. A comparison of two-level and multi-level modelling for cloud-based applications. In Gabriele Taentzer and Francis Bordeleau, editors, *Modelling Foundations and Applications - 11th European Conference, ECMFA 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-24, 2015. Proceedings*, pages 18–32. Springer International Publishing, 2015. pages 41

[204] Alessandro Rossini, Juan de Lara, Esther Guerra, Adrian Rutle, and Uwe Wolter. A formalisation of deep metamodelling. *Formal Aspects of Computing*, 26(6):1115–1152, 2014. pages 143

[205] Markus Scheidgen. Textual modelling embedded into graphical modelling. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture – Foundations and Applications: 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings*, pages 153–168. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. pages 68

[206] Christian Schneider. On Integrating Graphical and Textual Modeling. Master's thesis, Real-Time and Embedded Systems Group, Christian-Albrechts-Universität zu Kiel, 2011. pages 68

[207] Enrico Schnepel. Gengmf: Efficient editor development for large meta models using the graphical modeling framework. *Model Driven Software Engineer-ing-Transformations and Tools*, page 98, 2008. pages 234

[208] Ed Seidewitz. Uml with meaning: Executable modeling in foundational uml and the alf action language. *Ada Lett.*, 34(3):61–68, October 2014. pages 252

[209] Bran Selic. Using uml for modeling complex real-time systems. In Frank Mueller and Azer Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems: ACM SIGPLAN Workshop LCTES'98 Montreal, Canada, June 19–20, 1998 Proceedings*, pages 250–260. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. pages 4

[210] Bran Selic, Garth Gullekson, and Paul T Ward. *Real-time object-oriented modeling*. Wiley & Sons, 1994. pages 4

[211] Sirius. Project Website. `http://www.eclipse.org/sirius/`, 2014. pages 5, 235

[212] Riccardo Solmi. *Whole Platform*. PhD thesis, University of Bologna, 2005. pages 241

[213] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009. pages 162, 178

[214] Harald Störrle. On the impact of layout quality to understanding uml diagrams: diagram type and expertise. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 49–56. IEEE, 2012. pages 51

[215] Harald Störrle. On the impact of layout quality to understanding uml diagrams: Size matters. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems: 17th International Conference, MODELS 2014, Valencia, Spain, September 28 – October 3, 2014. Proceedings*, pages 518–534. Springer International Publishing, Cham, 2014. pages 51

[216] Eugene Syriani, Jeff Gray, and Hans Vangheluwe. Modeling a model transformation language. In Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholon Cohen, and Jorn Bettin, editors, *Domain Engineering: Product Lines, Languages, and Conceptual Models*, pages 211–237. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. pages 239

[217] Eugene Syriani and Hans Vangheluwe. A modular timed graph transformation language for simulation-based design. *Software & Systems Modeling*, 12(2):387–414, 2013. pages 239

[218] Eugene Syriani, Hans Vangheluwe, and Brian LaShomb. T-core: a framework for custom-built model transformation engines. *Software & Systems Modeling*, pages 1–29, 2013. pages 239

[219] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon V. Mierlo Mierlo, and Huseyin Ergin. Atompm: A web-based modeling environment. In *Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*, volume 1115, pages 21–25, 2013. pages 5, 238

[220] Tim Teitelbaum and Thomas Reps. The cornell program synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, September 1981. pages 66

[221] The Open Group. Archimate 3.0 specification. `http://pubs.opengroup.org/architecture/archimate3-doc/toc.html`, 2016. pages 18

[222] Tiger Project. Project Website. `http://www.user.tu-berlin.de/o.runge/tfs/projekte/tiger/`, 2015. pages 234

[223] Juha-Pekka Tolvanen. History and family tree of modeling languages. `http://www.metacase.com/blogs/jpt/blogView?entry=3575138300`, 2014. pages 1

[224] Juha-Pekka Tolvanen and Steven Kelly. Metaedit+: Defining and using integrated domain-specific modeling languages. In *Proceedings of*

the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, OOPSLA '09, pages 819–820, New York, NY, USA, 2009. ACM. pages 5, 16, 237

[225] Juha-Pekka Tolvanen, Risto Pohjonen, and Steven Kelly. Advanced tooling for domain-specific modeling: Metaedit+. In J. Sprinkle, J. Gray, M. Rossi, and JP Tolvanen, editors, *The 7th OOPSLA Workshop on Domain-Specific Modeling, Finland*, 2007. pages 238

[226] Dirk van der Linden and Henderik Alex Proper. On the accommodation of conceptual distinctions in conceptual modeling languages. In *Proceedings of the conference Modellierung 2014*, pages 17–32, 2014. pages 4

[227] Simon Van Mierlo, Bruno Barroca, Hans Vangheluwe, Eugene Syriani, and Thomas Kühne. Multi-level modelling in the modelverse. In *MULTI 2014 – Multi-Level Modelling*, volume 1286, pages 83–92, 2014. pages 3, 245

[228] Robbie Vanbrabant. *Google Guice: agile lightweight dependency injection framework*. Apress, 2008. pages 242

[229] Sander D. Vermolen, Guido Wachsmuth, and Eelco Visser. Reconstructing complex metamodel evolution. In *Proceedings of the 4th International Conference on Software Language Engineering*, SLE'11, pages 201–221, Berlin, Heidelberg, 2012. Springer-Verlag. pages 148

[230] Visual Modeling and Transformation System. Project Website. `https://www.aut.bme.hu/Pages/Research/VMTS/Introduction`, 2016. pages 234

[231] V. Viyović, M. Maksimović, and B. Perisić. Sirius: A rapid development of dsm graphical editor. In *Intelligent Engineering Systems (INES), 2014 18th International Conference on*, pages 233–238, 2014. pages 5, 16, 235

[232] Markus Voelter. Language and ide modularization and composition with mps. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering IV: International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011.*

Revised Papers, pages 383–430. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. pages 4, 242

[233] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards user-friendly projectional editors. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering: 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, pages 41–61. Springer International Publishing, 2014. pages 71

[234] Markus Voelter and Konstantin Solomatov. Language modularization and composition with projectional language workbenches illustrated with mps. *Software Language Engineering, SLE*, 16, 2010. pages 4, 242

[235] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. pages 33, 66

[236] Bernhard Volz and Stefan Jablonski. Towards an open meta modeling environment. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*, DSM '10, pages 17:1–17:6, New York, NY, USA, 2010. ACM. pages 18, 245

[237] Bernhard Volz, Stefan Zeising, and Stefan Jablonski. The open meta modeling environment. In *ICSE 2011 Workshop on Flexible Modeling Tools (FlexiTools 2011)*, 2011. pages 3, 245

[238] Jos B Warmer and Anneke G Kleppe. *The object constraint language: getting your models ready for MDA*. Addison-Wesley Professional, 2003. pages 113

[239] N. Wirth. Extended Backus-Naur Form (EBNF), ISO/EIC, 1996. pages 66

[240] XText. Project Website. `https://eclipse.org/Xtext/`, 2016. pages 241

[241] Luca Zamboni. *Getting Started with Simulink*. Packt Publishing, 2013. pages 51

[242] Steffen Zschaler, Dimitrios S. Kolovos, Nikolaos Drivalos, Richard F. Paige, and Awais Rashid. Domain-specific metamodelling languages for software language engineering. In Mark van den Brand, Dragan Gašević, and Jeff Gray, editors, *Software Language Engineering: Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers*, pages 334–353, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. pages 42