TECHNISCHE UNIVERSITÄT DRESDEN

DEPARTMENT OF COMPUTER SCIENCE
INSTITUTE FOR SOFTWARE AND MULTIMEDIA TECHNOLOGY
SOFTWARE TECHNOLOGY GROUP
PROF. DR. UWE ASSMANN

Großer Beleg

Design and Prototypical Implementation of a Pivot Model as Exchange Format for Models and Metamodels in a QVT/OCL Development Environment

Matthias Bräuer

Supervised by:

Dr.-Ing. Birgit Demuth

Task description

During the last years, the importance of domain-specific languages (DSL), which build on a simple meta-metamodel such as EMOF or EMF Ecore, has strongly increased. This development poses new challenges for the use of OCL as constraint and query language, because a standardized metamodel, such as the one of the UML, can no longer be assumed. In addition to that, OCL has become part of the model transformation language QVT (Query/View/Transformation) which also can be applied to different metamodels. An idea for managing the complexity is the design of a so-called Pivot Model as an exchange format for models and metamodels in an OCL/QVT development environment. Furthermore, a mechanism is required that either statically or dynamically describes respectively realizes the mapping between the Pivot Model and the metamodels to be supported.

The goal of the work is therefore the design of an adequate Pivot Model as well as a proposal for realizing the pivot principle including prototypical investigations. The pivot principle should facilitate the integration and reuse of components of the Dresden OCL Toolkit in a future OCL/QVT development environment.

To this end, the following partial tasks are to be solved:

- Study of current and relevant research literature
- Analysis of metamodel relationships between UML/MOF and OCL
- Requirements analysis regarding the future integration of the Dresden OCL Toolkit into the OCL/QVT development environment
- Design of the Pivot Model
- Research into concepts for realizing the pivot mechanism (e.g., composition technologies, “model weaving”, package merge in UML 2.0, model transformations)
- Experiments to prototypically implement the proposed mechanism (preferably within the bounds of the Eclipse Modeling Framework (EMF))

Acknowledgments

I would like to express my gratitude to my voluntary reviewers Swee Shean, Olli and Florian whose help in spotting weak parts, gaps in the logical flow, lacking explanations and errors in my English or slips of the pen has been sincerely appreciated. A special thanks goes out to Dr. Steffen Zschaler who provided lots of valuable feedback during the OCL "Stammtisch" meetings. Last not least, I would like to thank my supervisor, Dr. Birgit Demuth, for pointing me to interesting literature and helping with the overall organization and execution of this project.

Contents

1 Introduction

1.1 Motivation

Software engineering has come a long way since its first explicit mention in 1969 [NR68]. Continuously, new methodologies and paradigms have emerged, each time raising the abstraction bar further. From structured programming to object-oriented development to component-based systems, software engineers have strived to produce more reliable software faster and with less effort. However, the complexity of software has advanced equally rapidly resulting in ever increasing demands for even higher productivity in software development. To address these challenges, two interrelated research directions may be identified.

Firstly, modern approaches aim to involve experts in the respective application domain deeper into the development process. Traditionally, their participation often ends after requirements elicitation and resumes only during the final stages of acceptance testing. This seems odd, given that domain knowledge constitutes one of the central elements of every software system. Yet, it can be partly justified by the inherent complexity of tools and methodologies employed during the analysis, design and implementation phases. Thus, a central goal should be to equip domain experts with tools that are tailored to their domain of expertise.

The second aim of current research in software engineering is to "industrialize" the software engineering process. In other words, developing software should become more like building cars in a modern assembly line: timely, efficient and flexible in respect to customization demands by different clients. This stems from the observation that planning, designing, developing and maintaining a complex software system is still more of an art than a craft requiring significant experience and creativity. Today, software engineers are faced with a bewildering choice of frameworks, component technologies, application servers, operating systems as well as scripting and programming languages. Ideally, many of the more tedious and error-prone tasks such as manual coding or integration with the underlying platform should be automated, hidden away or abstracted from. Software engineers could then concentrate on the concrete functional requirements of a system rather than dealing with its low-level infrastructure. Additional benefits include increased quality, better reusablity and, ultimately, reduced *time to market (TTM)*, which increasingly has become the focal point of interest among product managers.

As noted in [SV06, p. 14], these goals are anything but new and represent "something like the IT industry's Holy Grail". However, throughout the last years many promising proposals have appeared that may indeed move the level of abstraction in software engineering one notch higher. One of these is *Model-Driven Software Development (MDSD)* which promotes models to first class artifacts in the software development process. Rather than relying on a single monolithic modeling language, MDSD introduces the idea of modeling a complex system using a variety of different *domain-specific languages (DSLs)*. Through domain-specific notations and specialized tools, MDSD aims to achieve the above-mentioned integration of domain experts into the analysis and design phases. Model transformation, code generation and the notion of an exchangeable platform are MDSD's answers to the challenges of automation and industrialization.

A key requirement for MDSD are formally specified, unambiguous, and precise models. To this end, a model written in a DSL may be enriched with expressions in a declarative, formally grounded constraint language specifying its semantics. However, defining a new constraint language for each DSL is neither practical nor feasible. As an answer, this report is going to examine how the *Object Constraint Language (OCL)* — the standard language defined by the *Object Management Group (OMG)* for querying UML models — can be applied to arbitrary DSLs. Since OCL has become the foundation of the model transformation language *QVT (Query/View/Transformation)*, the results of this work show up exciting new possibilities for the standards-based realization of the MDSD vision.

1.2 Aim and Scope

The aim of this report is to design a pivotal metamodel that can provide the necessary abstraction to evaluate OCL queries over instances of arbitrary domain-specific languages. Such a *Pivot Model* needs to be expressive enough to support all of the OCL language features. At the same time, it should be as simple as possible to allow for an easy mapping between the Pivot Model elements and the concepts of the DSLs that are to be integrated. To practically validate the pivot principle, a second goal of this project is to develop a flexible and extensible mechanism for defining and realizing this mapping. The prototypical implementation will use the *Eclipse Modeling Framework (EMF)* as its technological basis.

This work is done as part of ongoing research that aims to modernize and extend the functionality of the Dresden OCL2 Toolkit, a widely used OCL library and tool collection. Since the Toolkit's current infrastructure relies on an outdated model repository whose development is discontinued, significant re-engineering efforts are necessary. Naturally, these cannot be entirely achieved within the scope of a single report. In particular, this project does not provide a complete OCL engine based on the new architecture. Also, the time frame available did not suffice to exhaustively evaluate the proposed solution. Nevertheless, a thorough review of the literature and related work as well as a comprehensive documentation of design decisions strengthen the arguments made in this work. The first practical results look promising and should form a solid foundation for future work.

1.3 What does "Pivot Model" mean?

The term "pivot" in engineering usually refers to the center of a rotational movement. As noted in [Wen06b, p. 50], "pivot" is also used to describe an interpreter in simultaneous interpreting who translates from a less common language (e.g., Maltese) into an intermediate exchange language (e.g., English). The translated text then forms the basis for a translation into other languages. In Computer Science, previous adoptions of the term include the "pivot element" in the sorting algorithm *Quicksort*. Milanovic et al. [MGG$^+$06] introduce the notion of a "pivotal metamodel" for the *REWERSE Rule Markup Language (R2ML)* which they employ for a two-way mapping between the *Semantic Web Rule Language* in conjunction with the *Web Ontology Language (OWL/SWRL)* and UML along with OCL. This report follows their proposition by defining the "Pivot Model" as an intermediate metamodel used for aligning the metamodels of arbitrary domain-specific languages with that of OCL.

1.4 Organization of this Report

1.4.1 Chapter Structure

This document is structured as follows: Chapter 2 provides the necessary theoretical foundations and introduces the terminology used throughout the remainder of the report. This includes, in particular, a discussion of the concept of metamodeling and a review of relevant standards. Chapter 3 adds background information on tools and frameworks I have used for realizing the practical aspects of this work. In Chapter 4, I analyze the core problems and requirements of this project in detail and derive a conceptual framework that serves as a guidance for the remaining chapters. Chapter 5 thoroughly investigates related work and outlines the respective strengths and weaknesses. Additionally, I draw conclusions for my own work which is comprehensively presented in Chapter 6. Finally, Chapter 7 concludes on this work and provides directions for further research.

1.4.2 Typographical Conventions

To clearly distinguish semantically different elements and provide a visual hint to the reader, this report uses the following typographical conventions:

- *Italic script* highlights important key words, scientific terms and proper names of patterns, methods and tools carrying special meaning in the software engineering literature.
- Sans-serif font refers to elements contained in UML diagrams and other models if they represent an *abstract* concept rather than the implementation in a programming language.
- `Typewriter font` is used for code listings and the names of classes, methods and other code elements when referred to in the text.
- **Bold face** denotes table headers and items in enumerations.
- Blue text color represents hyperlinks referencing bibliography entries, sections, figures, etc. in the PDF version of the document.

2 Theoretical Foundations

This chapter reviews the key theoretical concepts, standards and technologies which this report builds on. Among others, this includes an introduction to the many acronyms prevalent in this field of study. The chapter starts with a comprehensive coverage of the concept of metamodeling, including its importance for the definition of domain-specific modeling languages. It continues by describing the four-layered meta hierarchy which forms the basis for most current approaches to model-driven software development. In this context, I identify two important foundational problems that need to be addressed to fulfill the aims of this report. Following is a brief outline of the motivation and expected benefits of model-driven software development and an overview of model transformation scenarios. Finally, I present a set of standards defined by the Object Management Group (OMG) that are relevant in the context of this project.

2.1 Metamodeling

2.1.1 Overview

Metamodeling is one of the most important foundations of *Model-Driven Software Development (MDSD)*. Since an underlying objective of this report is to investigate new methods for supporting MDSD, an introduction into the motivation and concepts of metamodeling becomes necessary. For instance, Stahl and Völter [SV06, p. 85] have identified the following MDSD challenges as being dependent on metamodeling:

- construction of domain-specific modeling languages
- model validation
- model-to-model transformations
- code generation
- tool integration

For this project, the aspects *domain-specific modeling languages* (Section 2.1.2), *model validation* (Section 2.3.4) and *model transformation* (Section 2.2.2) are of special interest.

However, to properly describe metamodeling itself, I have to briefly review the concept of models in general, with special regard to their use in software engineering.

Models

Models play an important role in most engineering disciplines. They are used in, e.g., architecture and civil engineering as well as mechanical and electrical engineering. The application of models in software development has a long tradition, too, with data modeling using the entity-relationship methodology [Bal00, pp. 224] being just one well-known example. With the rise of object-orientation during the 1990s and the development of the *Unified Modeling Language (UML)* [OMGd] the use of models in software engineering has become even more popular.

There is no mutually accepted definition of what a model actually is nor what its main purpose should be. In addition, there is a current shift in the way models are seen - from "simple sketches of design ideas" [Sei03] merely serving the purpose of documentation to first class software development artifacts. Especially model-driven approaches promote the unifying notion of "Everything is a model" [Béz05] in analogy to the object-oriented principle "Everything is an object".

For the purpose of this report, a less dogmatic view suffices. Following is a selection of definitions that I consider appropriate as a reference for the remainder of this work:

> "A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system." [BG01]
>
> "A model is an abstraction of a [..] system allowing predictions or inferences to be made." [Küh06]
>
> "A model is an abstract representation of a system's structure, function or behavior." [SV06]

The important observation here is that a model *describes* a system on a *higher* level of abstraction. This can be achieved by *categorizing* or *classifying* the different objects in the system rather than creating one model element for each system element. It should be noted, though, that the UML defines the concept of a *snapshot* [OMG05c, p. 55] to model the exact state of a system at a particular point in time. Yet, in this report, I only consider models that indeed *abstract* from the underlying system. This type of models has been called *type models* as opposed to *token models* [Küh06]. Even though abstracting, the model has to maintain enough expressiveness that all its statements about the system are true [Sei03]. Finally, a model needs to be written in a concise and non-ambiguous *language*. This simple relationship is summarized in Figure 2.1.

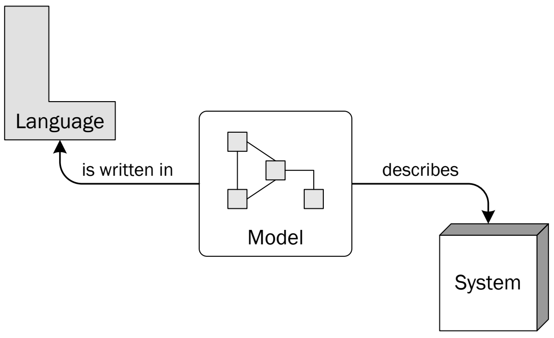


Figure 2.1: *Models and languages (adapted from [KWB03, p. 17])*

Modeling Languages

In principle, any language is suitable for creating models. The source code of a Java application, for instance, can be considered a (comprehensive, but complex) model of the actual running system. That is why, in line with [SV06, p. 18], I will clearly separate *programming languages* like Java from dedicated *modeling languages* as defined below. Also, textual descriptions of systems using natural language are not within the scope of this work. Modeling languages as referred to in this report possess a *human-readable, preferably visual notation* that may be complemented by *formal, declarative textual expressions.* A well-known example fitting this definition is the Unified Modeling Language (UML) in conjunction with the Object Constraint Language (OCL) (see Section 2.3.4).

Like computer languages in general, a modeling language is specified by its *syntax* and *semantics* [Bür05, p. 16]. The following distinctions are important:

Concrete Syntax The concrete syntax specifies the set of symbols and their arrangement in a valid representation of a model. More precisely, it defines what "a parser for the language accepts" [SV06, p. 57]. The graphical notation of the UML is an example for a concrete syntax. Note that there may be several different concrete syntaxes for a language.

Abstract Syntax The abstract syntax describes the structure of a modeling language, i.e. it specifies the concepts of the language and their relationships to each other. Thus, it abstracts from representation-specific aspects such as the spelling of keywords or the shape of graphical elements. For example, the concepts *Class*, *Operation* and *Association* are part of the abstract syntax of UML.

Static Semantics The static semantics are a set of wellformedness rules over the abstract syntax of a modeling language. They represent constraints that a model in that language has to fulfill in order to be valid. As an example, the UML specification [OMG05c, p. 82] disallows cyclic dependencies in inheritance trees by introducing the wellformedness rule that a Classifier cannot be a subclass of itself.

Dynamic Semantics The dynamic semantics define the meaning and behaviour of instantiated model elements. For instance, the UML Generalization element introduces the concept of type conformance. Thus, an instance of a specific Classifier which is related to another Classifier by generalization is also an instance of that general Classifier. Together, all UML semantics describe what has been called the "theory of UML class modeling" [Sei03]. For the purpose of this report, however, dynamic semantics are not relevant and I will not go into more detail.

Now, a question that naturally arises is: How can the different aspects of a modeling language be defined? To this end, MDSD rarely uses traditional language specification techniques such as context-free grammars and declarative or operational semantics [HKKR05, pp. 321]. Instead, models themselves are employed for this purpose.

Unfortunately, models are not capable of providing complete language specifications [Küh06]. In particular, the concrete syntax and dynamic semantics are difficult to define (though there are approaches proposing dedicated concrete-to-abstract-syntax mapping models [VKEH06]). As a practicable solution, the UML specification currently uses purely verbal explanations and examples in natural language to denote its concrete syntax and dynamic semantics. Unsurprisingly, this lack of mathematical rigour is one of the central points of criticism of researchers and practicians alike [HKKR05, p. 48]).

This brings me back to the start of this section since these "language definition models" introduced just now are called *metamodels*.

Metamodels

The Greek prefix "meta-" means *after* or *beyond* [MSUW04, p. 37]. As noted in [Küh06], "meta" is often used when "an operation is applied twice". Thus, a "meta-discussion" is a dicussion about how to conduct a discussion, "meta-learning" means to learn general learning strategies.

In MDSD, a metamodel is a "model of the modeling language." [MSUW04, p. 38]. More precisely, a metamodel describes the possible structure of models written in that language, i.e., the "constructs of [the language] and their relationships, as well as constraints and modeling rules" [SV06, p. 85]. Thus, a metamodel defines the abstract syntax and static semantics of a modeling language. Figure 2.2 shows an excerpt from the UML metamodel depicting a few basic class modeling constructs.

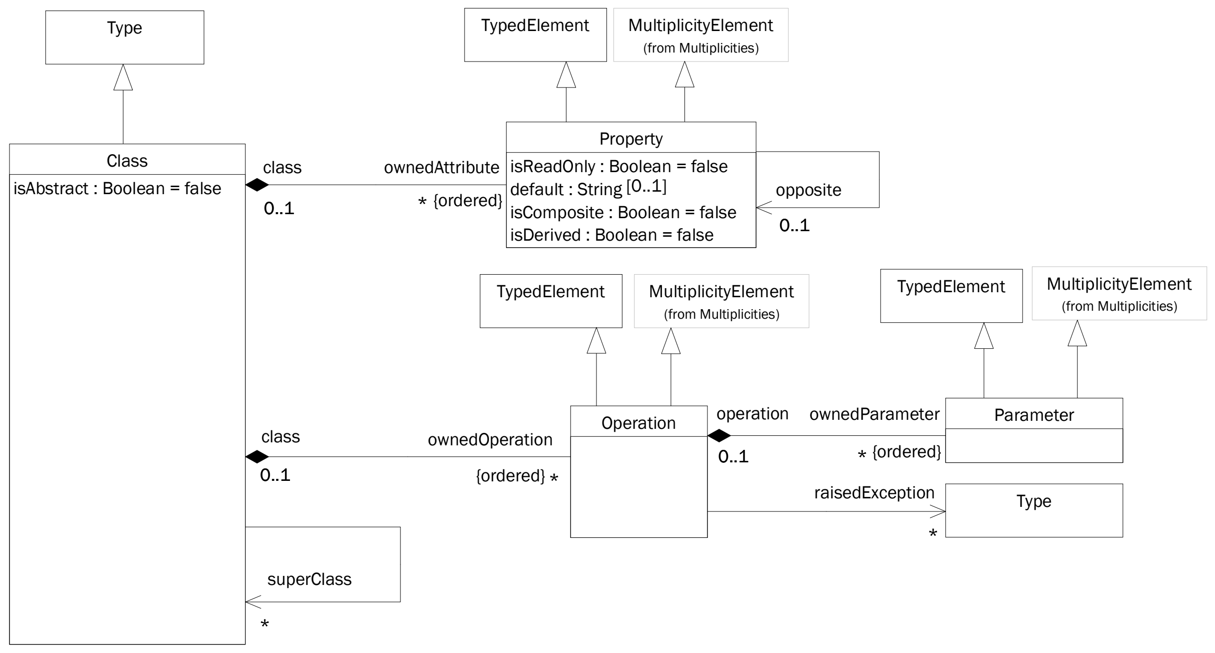


Figure 2.2: *Excerpt from the UML metamodel [OMGc]*

Obviously, a metamodel $MM(L)$ defining a modeling language L is at a higher abstraction level than a model M_L written in L. From a conceptual point of view, the elements of the metamodel are classifications of the different model elements — just like a model classifies the objects in a real system.

Of course, since a metamodel is a model itself, it needs to be written in a modeling language. This observation is illustrated in Figure 2.3. Now, if this language in turn is specified by a metamodel one speaks of a *meta-metamodel*. In theory, this abstraction sequence could be continued indefinitely but practice has shown that this does not yield any advantages (see Section 2.1.3).

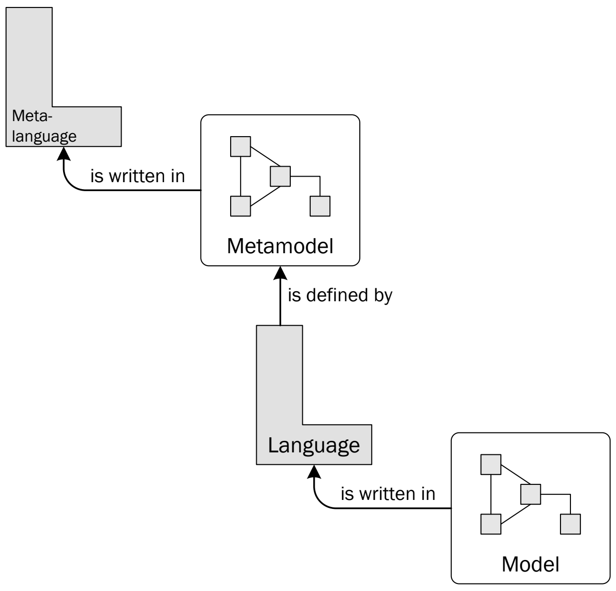


Figure 2.3: *Models, languages, metamodels, and metalanguages (adapted from [KWB03, p. 84])*

2.1.2 Domain-Specific Languages

A domain-specific language (DSL) is a language that is "tailored to a specific application domain" [MHS05]. A domain can be defined as an "area of knowledge [that] includes a set of concepts and terminology understood by practitioners in that area" [CE00, p. 34]. In contrast to general-purpose languages (GPLs) such as Java or C#, a DSL may therefore only express a limited set of concepts and is only suitable for a "specific class of problems" [Fow05]. The degree of "domain-specificness" varies between different languages. For instance, COBOL may be seen as a DSL for business applications or a full-featured third-generation programming language [MHS05]. Often, a DSL is said to capture a domain's *ontology* [GPvS02] (a term from the field of knowledge engineering [GPFLC04]).

Due to the use of domain-specific notations and a reduced amount of required programming expertise, DSLs offer a number of advantages over languages targeting a broader application scope. Summarizing [MHS05, Fow05], these are:

- substantial gains in expressiveness and ease of use
- possibilities for optimized tool support (editing, pretty-printing, debugging etc.)
- increased developer productivity
- reduced maintenance costs
- larger target audience (software developers, managers, domain experts)
- opportunities for domain-specific optimization and parallelization

The idea of domain-specific languages is not new. DSLs have played an important role in systems development for years. Table 2.1 lists some well-known representatives and their corresponding application domain. It is worth pointing out that the scope and complexity of DSLs varies greatly — from simple XML configuration files to full-blown visual notations [SV06, p. 144].

DSL	**Application Domain**
BNF	Syntax specification
Excel	Spreadsheets
HTML	Hypertext web pages
LaTeX	Typesetting
Make / Ant	Software building
MATLAB	Technical computing
SQL	Database queries and manipulation
VHDL	Hardware design

Table 2.1: *Examples of domain-specific languages (adapted from [MHS05])*

An important characteristic of software development using DSLs is that typically several DSLs are required to specify a complete application — each capturing one facet of the target system [WK06]. This approach is also called *language-oriented programming (LOP)* [Dmi04]. Using multiple, modularly composable DSLs yields several benefits such as increased reusability, scalability and fast feature turnover [CE00].

Recently, a convergence of model-driven software development and DSL engineering can be witnessed [BJKV06]. This stems from the observation that both approaches share the idea of language engineering as a means to describe systems. A number of proposals from industry and academia (Microsoft Software Factories [GS04], MetaCase MetaEdit+ [MC07], ISIS GME [LMB$^+$01], and Eclipse GMF [GT06], to name just a few well-known representatives) further advance this evolution. It appears that, to a large extent, this development has been triggered by the perceived inability of the UML to adequately specify the fine-grained aspects of heterogeneous systems [HKKR05, p. 49]. Certainly, the concept of small, specialized DSLs is appealing in comparison to the highly complex, monolithic UML Superstructure.

In brief, a DSL in model-driven engineering is defined as a modeling language whose abstract syntax solely consists of the concepts from a particular problem domain. Whereas traditional DSLs have been specified using a variety of methods (XML grammars, EBNF definitions etc.) model-driven approaches employ metamodeling. A metamodel defining a domain-specific language may thus be called a *domain definition metamodel* [BJKV06]. Finally, *domain-specific modeling (DSM)* denotes the activity of designing and building systems using one or several DSLs [Coo06].

Note that in some publications the terms modeling language and DSL are used interchangeably [SV06, p. 58]. However, this view blurs the distinction between general-purpose modeling languages such as the UML and light-weight, specialized ones. Hence, I will not refer to the UML as being a DSL in this work.

It is worth mentioning that the definition of a DSL is not complete without a concrete syntax and an execution semantics definition (though the executability requirement is controversely discussed [MHS05]). Typically, *editors* and *generators* are created for this purpose as soon as the abstract syntax of a DSL has been specified [Fow05]. Within the context of this report, these will mostly be visual model editors and code generation facilities. Tool support can significantly ease this task (see Chapter 3).

Summing up, Figure 2.4 illustrates the concepts introduced so far using a UML-like notation. Note that the dashed inheritance arrow conveys that an Abstract Syntax is *realized* by a Concrete Syntax.

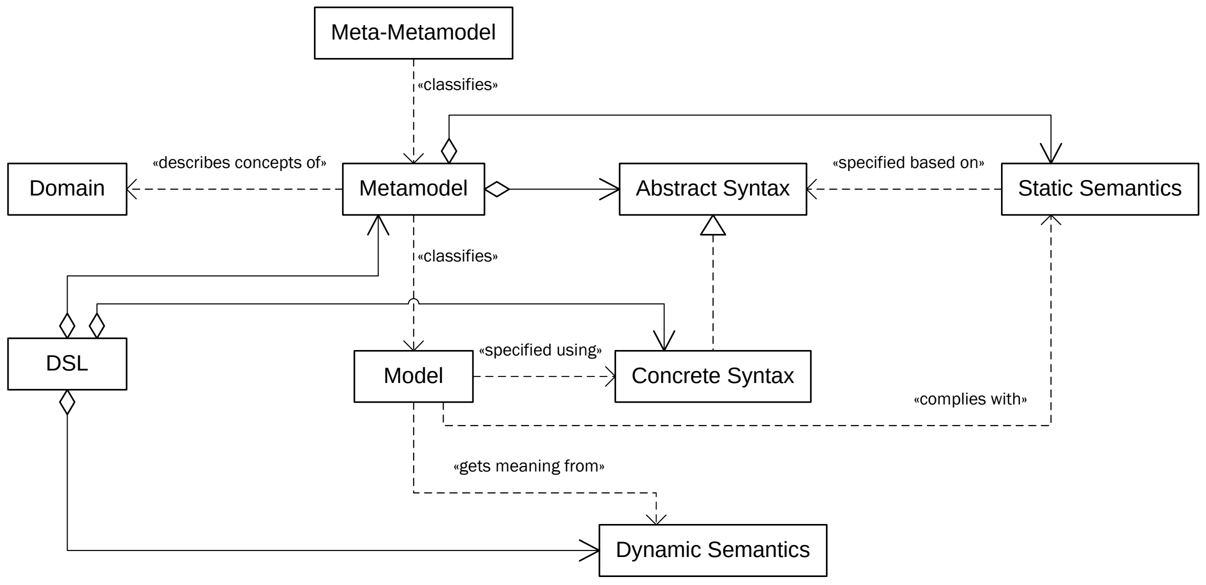


Figure 2.4: *Metamodeling, modeling and DSLs (adapted from [SV06, p. 56])*

2.1.3 The Four Meta Layers

The relation between models, metamodels and meta-metamodels as described in Section 2.1.1 has given rise to the idea of a *layered architecture* where each layer represents one level of abstraction. Higher layers thereby rely on increasingly smaller sets of concepts to describe the concepts in the layer below. At some point, there is no gain in defining another super layer because the constructs available are sufficient to define themselves. Thus, the modeling language defined by the metamodel can be used to model itself. At this level the metamodel is said to be *self-describing* [MSUW04, p. 43] and a *reflexive metamodel* [Sei03].

An important observation here is that the *meta* relationship is a relative one [SV06, p. 86]. In theory, it does not make sense to call a particular layer — such as the one with the model of the UML modeling language — a metamodel layer. This is because from a higher abstraction layer this "metamodel" simply appears as a "normal" model and, consequently, the higher layer should be labeled "metamodel" instead. Nonetheless, to avoid confusion a fixed designation has proven to be quite useful in practice. One concrete example is the *MOF metamodeling architecture* of the Object Management Group (OMG).

The MOF Metamodeling Architecture

MOF stands for *Meta Object Facility* [OMG06b]. Primarily, it is a meta-metalanguage for specifying metamodels. Besides the UML, other OMG metamodels have been defined using MOF, such as the *Common Warehouse Metamodel (CWM)* [OMG03a], the *Software Process Engineering Metamodel (SPEM)* [OMG05b] and the *Knowledge Discovery Metamodel (KDM)* [OMG06a]. As a reflexive metamodel, MOF itself is specified using MOF, too.

In addition, the MOF provides a framework for the management of metadata and a set of reflective services for creating, querying, manipulating and deleting metamodel elements [HKKR05, p. 328]. Declared design goal is to support interoperability between compliant metamodels and facilitate standardized model data exchange and model access [MSUW04, p. 44]. To this end, supporting standards such as *XML Metadata Interchange (XMI)* [OMG05a] and *Java Metadata Interfaces (JMI)* [Sun02] provide mappings to other technologies (XML files and Java-based model repositories, respectively).

In this report, only the metalanguage aspect of the MOF is of interest as it forms the basis for the four-layer metamodel hierarchy[1] on which the OMG suite of standards is built. Each meta layer in this architecture carries an explicit identifier (M0 to M3) and clearly specifies the kind of elements located on it. Lower layers are related to higher layers via an *instance-of* relationship. The top-most layer (MOF) is an instance of itself. Figure 2.5 illustrates the MOF hierarchy with a number of sample elements on each level. Summarizing [KWB03, pp. 85], the different meta layers can be characterized as follows:

Layer M0: The Instances This layer contains the run-time instances of model elements in the actual system. It is important to note that these instances manifest in various forms (objects inside object-oriented applications, rows in a database table, components in a distributed system), depending on the type of system being modeled. The key observation here is that the actual appearance of M0 elements is undefined. In software systems (and only those are within the scope of this report), they are just bits and bytes — software representations of real world items. I stress this point so strongly as the imprecise distinction of this matter in UML 1.x caused "endless semantic confusion" [RJB04, p. 403].

[1] Interestingly, the MOF specification explicitly refrains from setting an upper bound for the number of meta-levels. Actually, it is the UML specification that introduces the notion of four layers [OMG05c, p. 28].

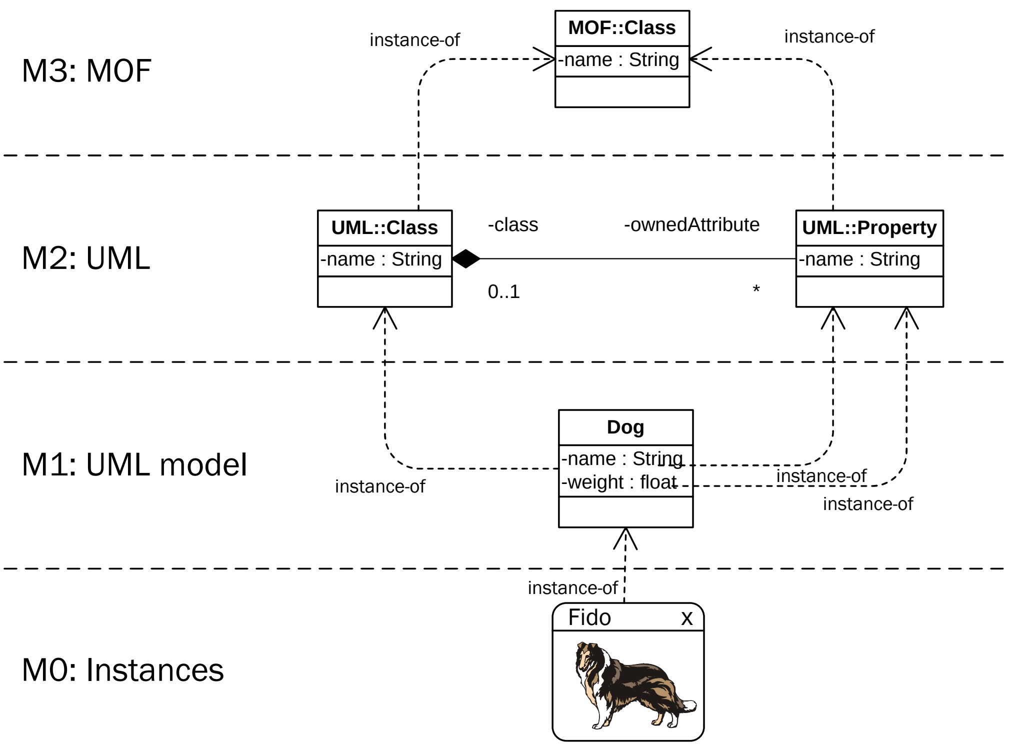


Figure 2.5: *The MOF metamodeling architecture*

Layer M1: The Model This layer contains the model elements describing the system as outlined in Section 2.1.1. Within the MOF architecture, a UML model resides on this layer.

Layer M2: The Metamodel This layer contains the elements of the metamodel, i.e. the concepts of the modeling language used to create the model on layer M1. This is where MOF-based metamodels such as the UML are located.

Layer M3: The Meta-Metamodel This layer only contains the MOF meta-metamodel as the OMG's standard M3 language.

Figure 2.6 shows the meta hierarchy that is the basis for the remainder of this work. It is a slightly more generalized version of the MOF architecture. A notable difference is that I do not restrict the meta-metalanguage to MOF. Indeed, partly due to lacking tool support for "true" MOF-based modeling I will employ another modeling framework for the definition of custom metamodels (see Chapter 3). Consequently, meta layer M2 may contain arbitrary DSLs and is not limited to the suite of MOF-based standards. Finally, I have renamed the M0 layer to *System* for better alignment with the terms introduced in Section 2.1.1.

The design of meta architectures is a non-trivial task and has been a focal point of research in metamodeling for several years. An attempt to classify metamodeling concepts and hierarchy design options can be found in [GH05]. Other publications [AK00, AK01, Béz05, Küh06] discuss certain shortcomings of the MOF architecture in greater detail. Among others, the number of meta layers, the kind of relationship between them and the representation of the "real world" are controversial issues [HKKR05, pp. 329]. Here, I will concentrate on the aspects that have immediate repercussions on the realization of my project goals. I call these the *ontological classification* and the *system instantiation* problems. Note that the full impact of these problems will only become obvious during the problem analysis presented in Chapter 4.

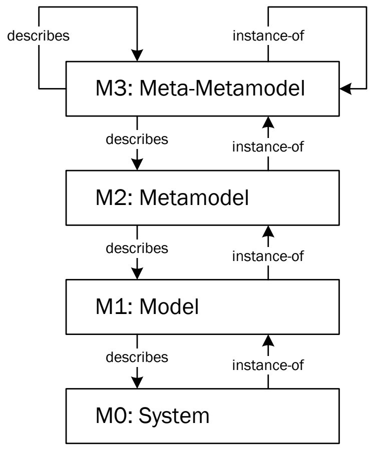


Figure 2.6: *A more generalized architecture (adapted from [SV06, p. 86])*

The Ontological Classification Problem

Atkinson and Kühne have identified two separate orthogonal dimensions of metamodeling [AK03]. The so-called *linguistic instantiation* is the relationship between elements on different levels of the classical four-layer meta hierarchy. This is metamodeling in the sense of language definition as presented in Section 2.1.1. However, within each layer there is an additional *ontological instantiation* relationship that associates elements according to a particular problem domain. Figure 2.7 illustrates this phenomenon with a simple example. As can be seen, the model element Dog is a linguistic instance of the meta element Class as well as an ontological instance of Species — another M1 element. One can also say that Dog *physically* instantiates Class and *logically* instantiates Species [AK02a]. Similarly, the element Fido (a model-level representation of the actual M0 instance) is an instance of both Object and Dog.

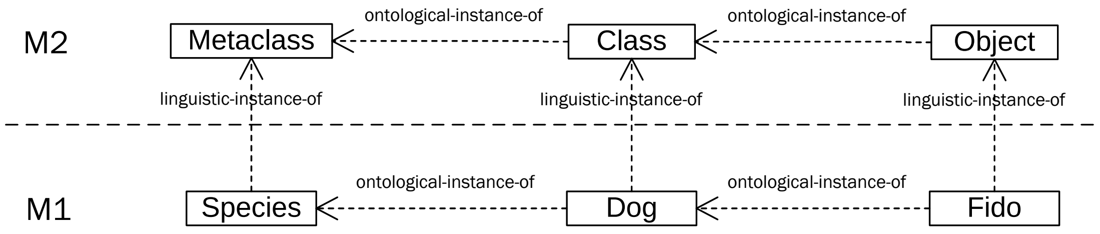


Figure 2.7: *Linguistic and ontological instantiation*

Although the MOF architecture has been criticized for not recognizing the two metamodeling dimensions explicitly [AK02b], it indeed provides a mechanism for expressing ontological relationships — using stereotypes and profiles. Figure 2.8 shows how the above-stated relationships can be expressed using actual UML metaclasses.[2] This "light-weight" metamodeling approach [HKKR05, p. 329] is controversial [AK00] but it yields one major advantage: Model querying languages like OCL (see Section 2.3.4) that are defined based on linguistic concepts (e.g, class, property and operation) can easily navigate on ontological entities (such as Dog). Through physical instantation the M1 element Dog possesses all necessary features.

[2] Contrary to depictions in many textbooks there is no dependency stereotype «instanceof» in UML to express ontological instantiation. In contrast, the stereotype «instantiate» may only be used between Classifiers indicating that "operations on the client create instances of the supplier." [OMG05d, p. 670]

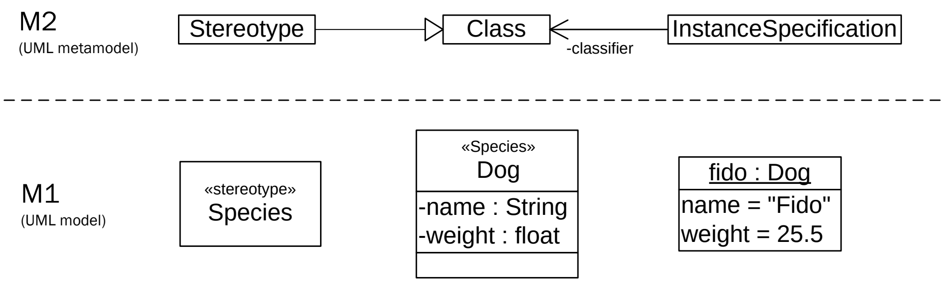


Figure 2.8: *Ontological instantiation using stereotypes in UML*

The approach I am following in this report (that is, defining ontological concepts on meta layer M2 in the form of a domain-specific language) creates significantly more challenges. To successfully use a language like OCL on models written in an arbitrary DSL, appropriate mappings between ontological and linguistic concepts are necessary.

The System Instantiation Problem

A closer look at the meta hierarchy in Figure 2.6 reveals that the *instance-of* relationship between the System and the Model layer is different from the other ones. Whereas the former crosses the boundary between "model space" and "system space", the latter all deal with instantiating models from meta-models. This issue has prompted proposals to alter the view of the meta hierarchy to a 3+1 architecture [BL97] and rename the connection between M0 and M1 to *represented-by* [BG01, Béz05].

A way of dealing with this problem is the notion of an "interpretation" of model elements with regard to the type of the system [Sei03]. An interpretation gives meaning to the model elements on layer M1. For instance, classes in a UML model may be interpreted as Java classes. The corresponding code can be generated by many standard UML tools. Then, the system level will be the realm of Java objects. If arbitrary DSLs are used to create the model, an interpretation needs to be provided by the writer of the DSL (e.g., in the form of a code generation definition).

A major implication in the context of this report is that creating new System layer instances requires knowledge of the model's interpretation. In Section 2.2.2, I will give an example for this situation in the context of model transformations. One aim of this report therefore is to find a mechanism to abstract from the way elements are instantiated.

2.2 Model-Driven Software Development

2.2.1 Overview

Model-Driven Software Development (MDSD), sometimes also called *Model-Driven Engineering (MDE)* or simplified to *Model-Driven Development (MDD)*, is an increasingly popular software development methodology that has received much attention over the last years.

The core idea of MDSD is to promote models to primary development artifacts that are equal to hand-written code. In fact, most of the code is supposed to be generated automatically. Implementations for different platforms can be based on the same models. Thus, models may even live longer than the code turning them into corporate assets for future reuse [MSUW04, p. 10]. Code generation may also include other artifacts like configuration files or deployment descriptors.

Instead of mere documentation, models in MDSD constitute a vital part of the system and stay consistent with the code over the entire software development lifecycle (SDLC). This contrasts heavily with earlier attempts at incorporating models into the software development process that often saw models quickly becoming outdated and out-of-sync with the actual implementation [KWB03, p. 2]. Addressing this paradigm shift, Stahl and Völter have emphasized the distinction between "model-driven" and "model-based" approaches [SV06, p. 15].

MDSD requires models to be complete, precise, and consistent specifications of a system with clearly-defined semantics. Then, unambiguous code generation becomes possible. Warmer and Kleppe have identified six *modeling maturity levels (MMLs)*, ranging from Level 0 (*No Specification*), which represents ad-hoc coding without any specification, up to Level 5 (*Models Only*), where no manual coding in a programming language is required any more. Current MDSD solutions aim at Level 4 (*Precise Models*) where large parts of the implementation are automatically generated and custom code is only required to fill the gaps. Architectural and design evolution always happens on the Model level and changes automatically propagate through to the code. This is an important difference to the roundtrip engineering approaches of the 1990s that usually altered the code directly to reflect adjustments of an initial model. Diagrams then merely served as a visualization of the updated design.

Among the main promises of model-driven software development are:

- increased productivity through automation of software production
- improved manageability of complexity through higher level of abstraction
- increased quality and reliability
- better maintainability and documentation
- easy portability to different platforms

In addition, MDSD can be the enabling technology for approaches such as *Feature-Oriented Programming (FOP)* [BBGN01] and *Product Line Engineering (PLE)* [CN01] that seek to model variability in software using so-called feature models. MDSD also eases the isolation of cross-cutting implementation aspects such as transactions, persistence, or authentication, and thus facilitates *Aspect-Oriented Software Development (AOSD)* [KLM$^+$97]. A more detailed discussion of the relationship between MDSD and AOSD can be found in [Völ05].

2.2.2 Model Transformations

Model transformations represent one of the cornerstones of Model-Driven Software Development. They can be defined as "sets of rules describing how models that conform to a meta-model are to be expressed in models that conform to a second (not necessarily different) meta-model." [KPP06c] In the context of the OMG *Model-Driven Architecture (MDA)* (see the next section) they are also called *Mappings* [MSUW04].

Possible application areas include the transformation into models with lower abstraction level, the integration of models written in different modeling languages (see Section 4.3.3), and the generation of code artifacts. Czarnecki and Helsen provide a thorough classification of model transformation approaches in [CH06]. Intensive treatments of the topic can also be found in [Wen06b] and [Pöt06]. Requirements for model transformation engines are comprehensively discussed in, e.g., [KWB03, p. 73] and [SV06, p. 204]. Here, my main focus lies on the relationship between model transformations and the meta hierachy introduced in Section 2.1.3.

Kurtev and van den Berg have identified several scenarios for model transformations [KvdB04]. An excerpt of their findings is shown in Figure 2.9. The left-hand side illustrates a typical model-to-model transformation performed on the Model layer (M1). The right-hand side, however, introduces the novel notion of transformations on the System level (M0). Here, a relational database is filled with data from a concrete XML document, based on mappings defined between the DTD (Document Type Definition) of the XML file and the database schema. The example demonstrates a use case that currently has to be implemented manually, e.g., using an XML parser and a program written in a general-purpose language like Java. The need for similar transformations over data has triggered the development of several specialized data query and transformation languages (e.g, XPath [W3C07a] and XLST [W3C07b] in the XML domain). Obviously, a common metamodel-based language for transforming heterogeneous data sources is desirable.

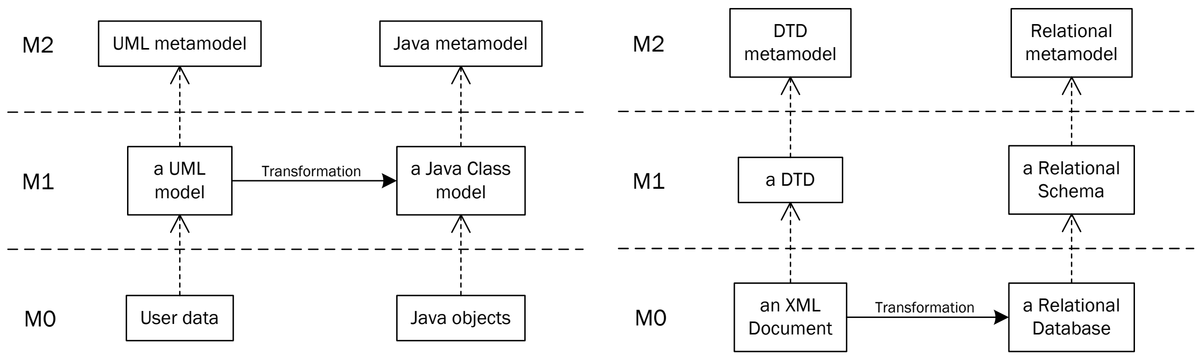


Figure 2.9: *Model transformation scenarios*

2.3 The OMG Standards

The Object Management Group (OMG) [OMGa] is an international industry consortium of more than 800 companies dedicated to the standardization of enterprise integration technologies. One of its main goals is to foster the interoperability and portability of software systems. It is probably most well-known for its specifications of middleware platforms (e.g., CORBA — Common Object Request Broker Architecture) and (meta-) modeling languages (e.g., MOF and UML). In this section, I will have a look at the set of standards relevant in the context of this report. The order of the presentation is such that each sub-section builds on the concepts explained in the previous one. I start by describing the "Package Merge" mechanism as a prerequisite for the modularization of MOF-based modeling languages and finish with a brief overview of the MDA framework, the OMG's contribution to the ongoing discussion about model-driven software development.

2.3.1 Package Merge

Package Merge is a novel metamodel definition technique introduced with UML 2.0. It is heavily used in the UML Superstructure Specification [OMG05d] to partition the language into four increasingly expressive compliance levels [HKKR05, p. 316]. Package Merge is intended to ensure the backwards compatibility of UML tools with XMI serializations from lower compliance levels. The effectiveness of this approach has been questioned, though [ZDD06].

Package Merge is modeled as a directed relationship between two packages using a dependency arrow with the stereotype «`merge`». Figure 2.10 shows a simple example.

In brief, the semantics of the Package Merge operation are as follows: The elements of the *merged package* are redefined in the *receiving package*. Thus, the effective package contents of the

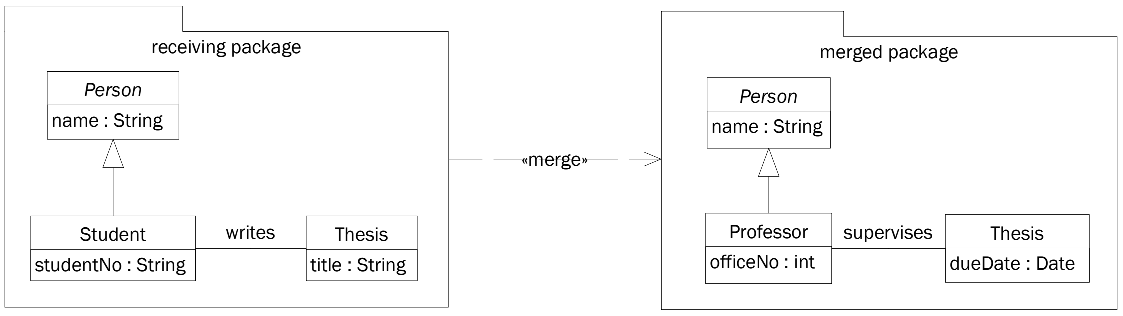


Figure 2.10: *An example for Package Merge*

resulting package is the union of the elements from both packages [HKKR05, p. 128]. Elements that exist both in the merged package and the receiving package are merged into one element. To determine candidates for a merge, by default the name and the metatype of the element are considered. Special rules exist for associations, operations and relationships. Figure 2.11 describes the result of the merge depicted in Figure 2.10.

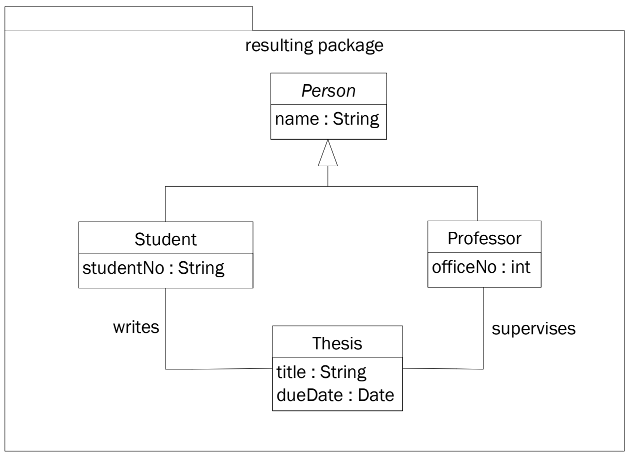


Figure 2.11: *The implicit result of the Package Merge*

Note that the package resulting from the merge is implicit. Conceptually, there is "no [semantic] difference between a model with explicit package merges, and a model in which all the merges have been performed" [OMG05c, p. 158]. In the UML specification, most often an empty base package merges several other packages to define a certain set of concepts.

2.3.2 The Common Core of UML and MOF

Due to different development histories, the 1.x versions of UML and MOF were structurally quite different. Although both target object-oriented class structures, many fine-grained implementation details varied [Ock03]. With the release of UML 2.0 and MOF 2.0, the two specifications have been aligned to share a common core of concepts. This minimal set of object-oriented language elements is called *Infrastructure* [HKKR05, p. 324] and already constitutes a meta-metalanguage that is expressive enough to specifiy a wide range of metamodels. Additionally, it provides a rich and precisely defined set of meta elements to be reused in other metamodels on the same metalevel. The relationships between UML, MOF and the Infrastructure are illustrated in Figure 2.12.

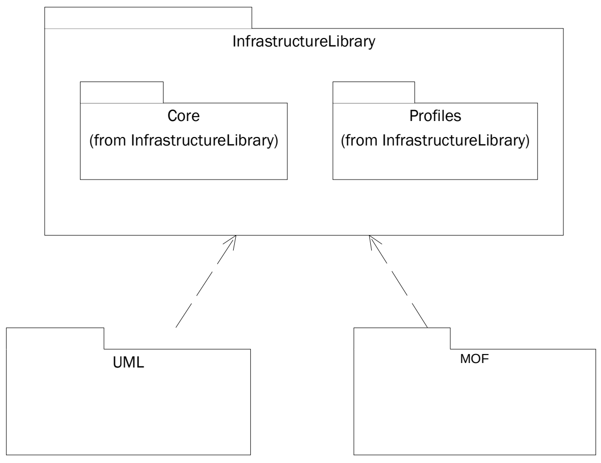


Figure 2.12: *UML, MOF and the Infrastructure Library*

The Infrastructure is represented by the package InfrastructureLibrary. Two subpackages Core and Profiles define the core concepts for modeling and the profile extension mechanism to define new UML-based languages, respectively [OMG05c, p. 12]. The Core package is further subdivided into the packages PrimitiveTypes, Abstractions, Basic and Constructs. Of particular importance is the Basic package because it forms the foundation for the *Essential MOF (EMOF)* dialect of the MOF meta-metalanguage (see next section). Figure 2.13 gives an overview of all elements in Core::Basic and their relationships.

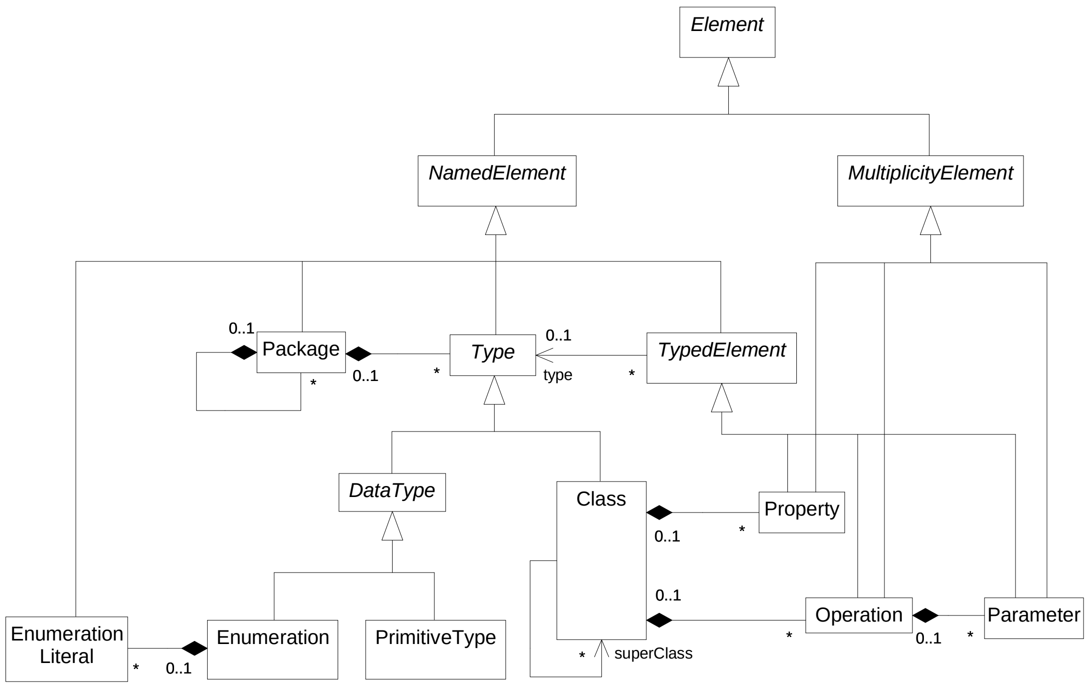


Figure 2.13: *The elements of the Core::Basic package*

2.3.3 Essential MOF

In an attempt to modularize the language definition, ease implementation and maximize interoperability, the MOF 2.0 specification defines a "kernel metamodeling capability" called *Essential MOF (EMOF)* [OMG06b, p. 41]. It represents the subset of *Complete MOF (CMOF)* that "closely corresponds to the facilities found in OOPLs and XML" [OMG06b, p. 43]. The explicit specification of EMOF as a simple language for defining simple metamodels is motivated by the observation that many metamodels do not require the expressive power provided by CMOF. Experience from parallel developments, such as the *Ecore* metamodel of the *Eclipse Modeling Framework (EMF)* (see Section 3.2), has significantly influenced this decision.

EMOF merges the Core::Basic package from the UML Infrastructure as well as additional MOF packages which provide the metadata management facilities mentioned in Section 2.1.3. This is shown in Figure 2.14. Note that this model has to be created in CMOF, because Package Merge is not supported by Core::Basic. However, the resulting model (i.e., after all merges have been performed) can be fully specified using EMOF concepts alone. Consequently, EMOF is an instance of itself and a reflexive metamodel (see Section 2.1.3).

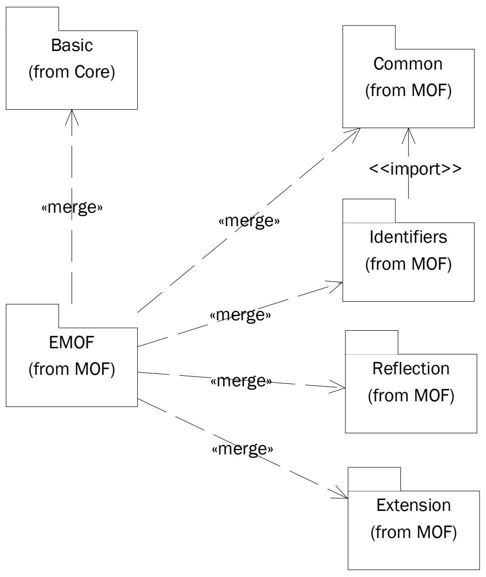


Figure 2.14: *Definition of EMOF using Package Merge*

2.3.4 The Object Constraint Language

Usually, the expressiveness of a visual modeling language like UML does not suffice to achieve the level of modeling precision required by MDSD (see Section 2.2.1). In particular, the definition of constraints restricting the set of valid model instances is not supported. In recognition of this problem, the OMG has introduced the *Object Constraint Language (OCL)* as a "standard 'add-on' to the Unified Modeling Language" [WK03]. OCL is a textual, side-effect-free, and declarative language for the specification of validity rules over models in MOF-based languages. Typical usages are the definition of class invariants as well as pre- and post-conditions of operations.

An important observation here is that OCL constraints are, in theory, modeling language-independent [SV06, p. 97]. In particular, they can enrich both metamodels (M2) and models (M1). Constraints on the Metamodel layer define the static semantics of a modeling language (see Section 2.1.1) and are usually called *wellformedness rules*. They are especially important for the precise definition of domain-specific languages in the MDA framework [Gog01, GNR04]. Note that a constraint always affects the instances of the constrained element on a lower metalevel, i.e., wellformedness rules for a DSL are validated on models written in that DSL.

The second release of OCL [OMG06c] introduced major improvements to the language. Instead of a context-free grammar, OCL 2.0 is formally based on a MOF metamodel. The OCL Standard Library, a collection of predefined types and operations on these types, can now properly be placed on the M1 Layer. Figure 2.15 illustrates the mapping between concrete and abstract syntax and the relationship to the Standard Library.

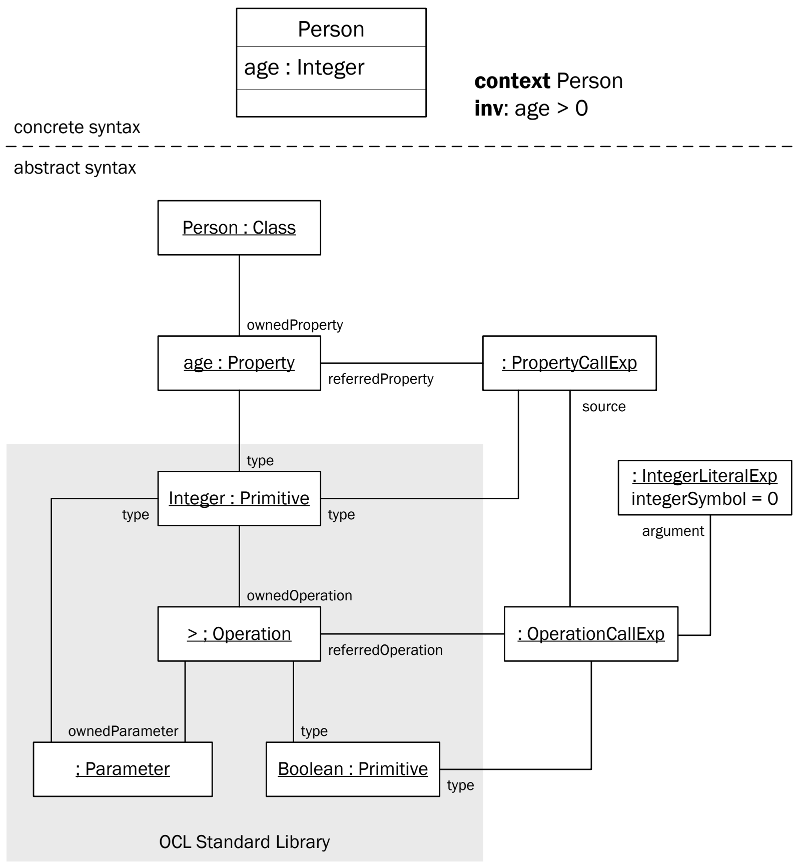


Figure 2.15: *Concrete and abstract syntax of an OCL expression (adapted from [Ock03, p. 13])*

Note that the OCL metamodel is precisely specified using OCL itself. This applies to wellformedness rules, the abstract syntax mapping and the definition of additional operations on UML metaclasses. As should be clear by now, the corresponding OCL expressions instantiate the OCL metaclasses on a higher metalayer. Obviously, an initial implementation of an OCL engine demands a bootstrapping process. In this report, I simply use Java for this purpose.

In addition to the new formal grounding, OCL 2.0 has matured “from a constraint language to a full query language for object-oriented models” [HZ04]. Earlier works had already recognized this potential [GR98]. Through the introduction of a tuple type, OCL 2.0 now has the expressive power of queries formed in Relational Algebra [AB01]. As a result, new usage types (in addition to those mentioned above) have been defined. These include initial and derived values of properties, operation body expressions and the definition of new operations and properties on classifiers.

2.3.5 Essential OCL

Essential OCL is the "minimal OCL required to work with EMOF" [OMG06c, p. 171] (Figure 2.16). It is structually identical to *Basic OCL* which exposes the OCL concepts required to work with Core::Basic. This is not surprising since EMOF is built from Core::Basic (see Section 2.3.3). Essential OCL is motivated by the same considerations as EMOF, namely, providing a simple query and constraint language for simple metamodels. Since this already represents an important step towards independence from the UML metamodel and application to arbitrary domain-specific langauges, this report bases its theoretical and practical investigations on the Essential OCL definition.

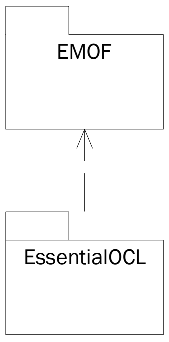


Figure 2.16: *Essential OCL depends on EMOF*

Essential OCL is currently not very well-defined. The specification simply states that class descriptions and wellformedness rules are to be reinterpreted for Core::Basic. Additionally, since Core::Basic does not know the notion of Classifier, all Classifier references have been adapted to reference the Type metaclass instead. Figure 2.17 shows the updated OCL type system resulting from this reinterpretation.

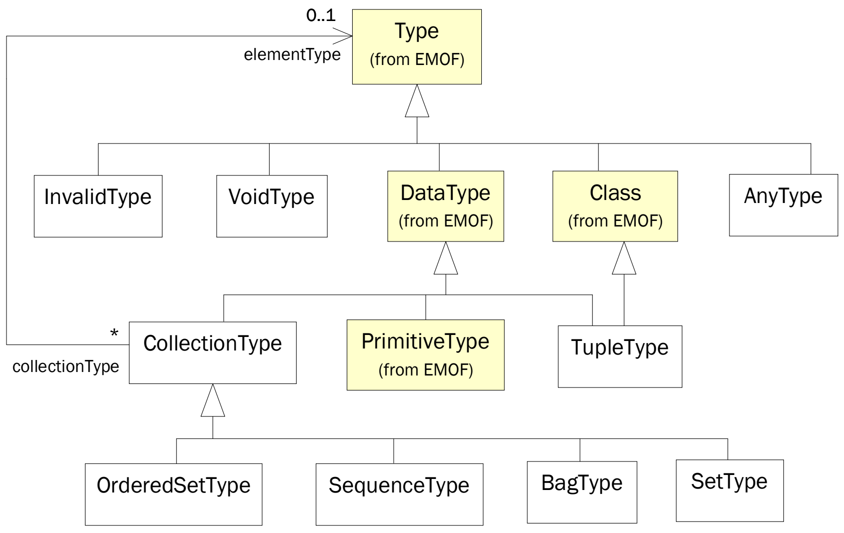


Figure 2.17: *The Types Package of Essential OCL*

Unfortunately, this yields two significant disadvantages. Firstly, since Types in Core::Basic may not possess operations, instances of the special OCL types AnyType, VoidType, and InvalidType in the OCL Standard Library cannot contain their predefined operations. The specification recommends the workaround of providing an additional Class instance with the same name that provides those operations. Similarly, Datatypes do not include the reference to Property. To enable TupleTypes to define attributes as in complete OCL (i.e., as Property instances), TupleType additionally inherits from Class. Without a doubt, these adjustments represent a feasible, albeit cumbersome approach. In this report, I will aim to find a better solution to this problem.

2.3.6 Query / View / Transformation

Query / View / Transformation (QVT) is the OMG answer to the problem of model transformations introduced in Section 2.2.2. It is a complex language and a discussion is outside the scope of this report. An overview of the language architecture, a review of existing implementations and example model transformations realized in QVT can be found in [SV06, p. 203] and [Wen06b]. In the context of this work, it suffices to note that QVT uses OCL (or rather extensions of it) for querying models. Hence, any attempt to realize a QVT engine has to provide a solid OCL implementation as its base first. In the remainer of this report, my main focus will lie on the discussion of a flexible design for OCL integration with regard to domain-specific languages. Yet, the ultimate goal of this and following research is its application in the area of model transformations. Consequently, some of the results presented in Chapter 6 have been developed in the light of this long-term objective.

2.3.7 The OMG MDA initiative

In 2001, the Object Management Group (OMG) adopted the MDA framework. MDA stands for *Model-Driven Architecture* and is, in the OMG's own words, an "approach to using models in software development" [OMG03b]. From another point of view, MDA may be seen as a specialization of MDSD based on OMG standards [SV06, p. 63]. For instance, the MDA framework assumes MOF as the meta-metamodel for specifying domain-specific languages. An alternative and (due to lacking tool support for metamodeling) viable approach is the extension of UML with profiles and stereotypes. Of the MDSD goals, the MDA particularly focuses on interoperability and portability, i.e., platform independence. To this end, it defines the concepts of *Platform Independent Model (PIM)* and *Platform Specific Model (PSM)*. PIMs are converted to PSMs via model transformations..

3 Tools and Technology

In the previous chapter, I provided the necessary theoretical background on concepts and standards important in the context of this report. Continuing from there, I will now introduce the practical technologies that were the basis for the prototypical implementation developed in the course of this project. Since I have realized the entire functionality as a set of Eclipse plug-ins, I will give an overview of the Eclipse platform as a runtime container for user extensions. In addition, I will briefly describe the (meta-) modeling capabilities and code generation facilities of the Eclipse Modeling Framework (EMF) as well as the Ecore metamodel it builds on. It is worth highlighting that the information presented here lays out the foundation for most of the examples and practical investigations contained in subsequent chapters.

3.1 The Eclipse platform

Eclipse is an “open development platform comprised of extensible frameworks, tools and runtimes for building [..] software across the lifecycle” [Ecl]. Originally introduced by IBM as a powerful Java IDE in 2001, the Eclipse project now hosts development environments for many other languages and technologies as well as frameworks for a wide variety of application areas, including web development [WTP], testing [TPT], reporting [BIR], and modeling [MDTa].

Fundamentally, Eclipse is a collection of plug-ins built around a very small runtime core. The platform readily supports many common features in modern user interfaces, such as wizards, editors and views, facilitating rapid development of rich client desktop application. Since the example in Section 4.1 will be based on the Eclipse plug-in mechanism, I provide a brief discussion here. A comprehensive coverage of the entire platform architecture and functionality can be found, e.g., in [CR06].

The extensible design of the Eclipse platform is centered around the concept of *extension points* which can be offered by plug-ins. The Eclipse Help System defines extension points as “well-defined places where other plug-ins can add functionality”. Thus, of the two goals of object-oriented framework design, Eclipse focuses on *extensibility* rather than *variability* [Aßm06]. Many extension points are already defined by the platform plug-ins (e.g„ to contribute new editors, menus or toolbar items), but custom plug-ins may add their own to provide extensible services.

Extension points are precisely specified with an XML Schema. Each extension point may define several attributes where each attribute can either denote a String or Boolean value, the path to a resource (e.g., an icon) or the name of a Java class. Plug-ins that wish to add custom functionality need to declare an *extension* for an extension point. An extension is defined via an XML fragment that conforms to the extension point’s XML schema. Conceptually, an extension is an *instance* of an extension point providing concrete values for its attributes. For example, an extension may define the text and icon for a new toolbar button and provide an implementation of the `IActionDelegate` interface that performs the associated action. An extension which contributes behavior by defining a Java class attribute is called *executable extension.*

3.2 The Eclipse Modeling Framework

The *Eclipse Modeling Framework (EMF)* [EMF] is a "powerful framework and code generation facility for building Java applications based on simple model definitions" [BSM+03]. Its goal is to enable data integration and interoperability for tools based on Eclipse as well as other platforms. To this end, it provides an abstraction layer over the Eclipse plug-in concept introduced in the previous section. EMF features metamodeling and code generation capabilities that strongly resemble the MOF-to-Java mappings defined by JMI (see Section 2.1.3). Yet, in contrast to the OMG suite of standards, EMF employs its own meta-metamodel called *Ecore* that is akin to, but not identical with MOF (see Figure 3.1). Recently and mainly due to the introduction of EMOF, a convergence of the two worlds can be witnessed, though. An assessment of the relationship between EMF and OMG standards is presented in [Gro06].

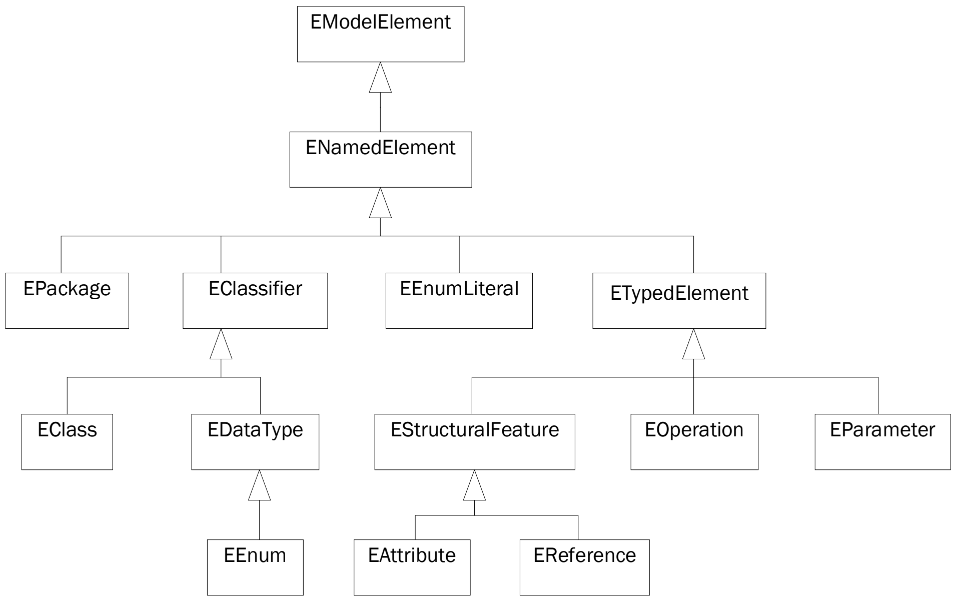


Figure 3.1: *The Ecore metamodel of the Eclipse Modeling Framework*

EMF now has a large user community and is the basis for many open-source and commercial projects. To a large extent, this popularity stems from the good integration with the ubiquitous Eclipse platform. Also, in contrast to JMI, EMF generates simple and easy-to-use Java code that does not depend on a complex repository infrastructure. From release 2.2 on, it is even possible to flexibly adapt the entire code generation process. Another driving factor for the success of EMF certainly is its comprehensive tool support. For instance, the framework allows to automatically generate a highly customizable editor for a modeling language specified in Ecore. Validation of Ecore metamodels via OCL or hand-written constraints is supported as well. Finally, in conjunction with the Graphical Editing Framework (GEF) [GEF] and the Graphical Modeling Framework (GMF) [GMF], building visual editors for domain-specific modeling languages becomes significantly easier. All these features pave the way for an EMF-based language workbench [Fow05] as introduced in Section 2.1.2.

4 Problem Analysis

In Section 2.1.2, I highlighted the growing trend towards small and specialized metamodels in favour of monolithic, general-purpose modeling languages. This development encompasses both the Meta-metamodel and the Metamodel level. In Section 2.1.2, I have already given a brief overview of the possibilities for DSLs on the Metamodel layer. By far not as many meta-metalanguages exist, but popular examples are EMF Ecore, which I introduced in the previous chapter, and KM3 [BJ06], the language used to define the DSLs of the AMMA tool suite [AMM]. This has significant impacts on the way model querying languages like OCL have to be implemented. Whereas traditional approaches focused solely on an integration with UML or, to a lesser degree, with MOF, this chapter will analyze the implications of applying OCL to instances of arbitrary domain-specific languages.

After presenting a motivational example, I will categorize the usage scenarios for OCL along two orthogonal dimensions. This classification is then extended by a conceptual research framework which will serve as a guideline for the remainder of the report. This framework represents one of the major contributions of my work as it allows to easily compare and evaluate existing approaches to integrating OCL with multiple metamodels. In this context, I also provide a thorough analysis of different model composition mechanisms highlighting their respective advantages and disadvantages. The chapter concludes with a concise requirements definition and an outlook on the benefits of the Pivot Model concept that is the focal point of research in this report.

4.1 A Motivational Example

Section 3.2 introduced the Ecore metamodel that is used to define EMF models. Ecore is an example for a DSL on the meta-metalevel. It only provides a subset of the concepts available in more complex metalanguages like MOF, but for most applications its expressiveness suffices. To illustrate the use of OCL for models and metamodels that are not based on the OMG stack of standards, I have devised a simple DSL modeled in Ecore. The DSL captures the concepts of the Eclipse platform's plugin mechanism and is accordingly coined *Plugin Modeling Language (PML)*. Its metamodel is shown in Figure 4.1.

4.1.1 The Plugin Modeling Language

PML models the containment relations between Eclipse features and associated plugins. It also allows specifying extension points and services offered by these plugins. Each extension point is assumed to declare an executable extension and, therefore, references a Java type that contributing extensions must conform to. Java types are identified by their fully-qualified class name. Note that this DSL is far from being complete. To keep the metamodel simple, I have left out attributes which are not strictly required (such as human-readable extension point names or descriptions). Also, I do not support the concept of additional extension point attributes.

As outlined in Section 2.1.1, a modeling language may have an arbitrary concrete syntax as long as an unambiguous mapping to the abstract syntax exists. Figure 4.2 exemplifies a visual

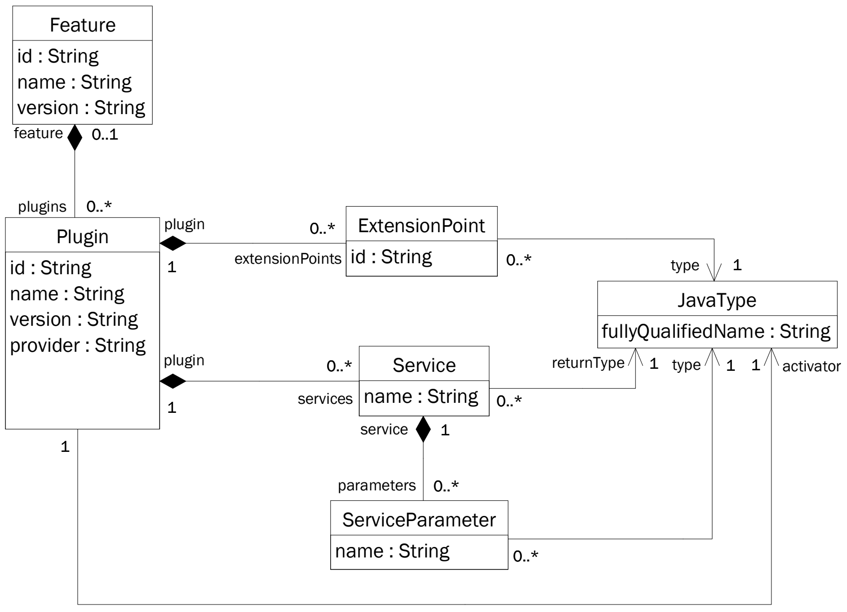


Figure 4.1: *Metamodel of PML*

syntax in a model of the Eclipse RCP (Rich Client Platform) feature including the definition of the Eclipse UI plugin and some of its extension points. Language workbenches, such as the Eclipse Graphical Modeling Framework (GMF) [GMF], may significantly ease creating a graphical editor for this concrete syntax from an abstract syntax model.

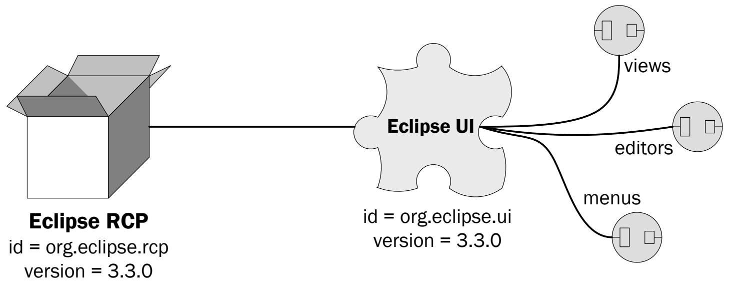


Figure 4.2: *A PML model*

4.1.2 Adding OCL expressions to Ecore and PML models

In model-driven software development, models need to be precisely specified (see Section 2.2.1). This includes, among others, adding validity constraints and defining derived model properties and operations. As discussed in Section 2.2.2, a query language for model transformations is desirable as well. The Object Constraint Language is an obvious candidate to meet these requirements.

As an example, consider the OCL constraints shown in Listing 4.1. They represent *well-formedness rules* over PML models and, thus, are *defined* on the Metamodel layer (M2). As described in Section 2.1.1, the abstract syntax representation of these expressions consists of instances of *concepts* defined on the Meta-Metamodel level. In this example, these are instantiated elements of the OCL as well as the Ecore metamodel. As a result, the instance of OCL::Expressions::PropertyCallExp that is created for the expression `self.id` has to reference the EAttribute representing the `id` attribute. Yet, neither PML nor its metalanguage Ecore are specified using UML or MOF, respectively.

```
-- a Plugin must have a valid id
context Plugin
inv: self.id->notEmpty()

-- all Plugins in a Feature must be distinct
context Feature
inv: self.plugins->isUnique(plugin | plugin.id)
```

Listing 4.1: *PML wellformedness rules expressed in OCL*

To make this actually work, OCL would have to be integrated with the Ecore metamodel. This is certainly possible [MDTb], especially given the great similarity between the Ecore and the EMOF metamodels. Nonetheless, it requires a certain effort which, unfortunately, has to be repeated for every new DSL. This leads to a proliferation of structurally identical OCL metamodels that differ only in their binding to a particular metamodel.

As an example for this problem, consider the invariants in Listing 4.2 that refer to the PML model introduced in the previous section. Obviously, these expressions are defined on the model level (M1). Each plug-in extension point is here interpreted as a multi-valued attribute of its owning plug-in that represents the list of extensions registered for that point. Note that the semantics of an OCL query on elements of an arbitrary DSL may not be intuitive. For more complex DSLs (especially those that bear little resemblance to the UML metamodel), several valid interpretations may be possible. Thus, a binding of OCL with a DSL usually requires a precise mapping definition which has to be provided by the implementor of the binding.

```
context EclipseUI

-- there must be at least one view extension
inv: self.views->notEmpty()

-- extension must implement IViewPart
inv: self.views.isKindOf(org.eclipse.ui.IViewPart)
```

Listing 4.2: *OCL constraints on a PML model*

The two examples provided so far also illustrate the equally important problem of expression evaluation. The expressions of Listing 4.1 are *executed* on meta layer M1, i.e., on actual PML models accessible via the EMF infrastructure. This is fundamentally different from the second case which specifies constraints for an actual running system. In the example, this is an Eclipse workbench. Clearly, rewriting an OCL execution engine for each OCL binding is undesirable.

To sum up, not only do different DSLs require their own OCL binding, but some may even be interpreted in several ways. In addition, OCL expressions may require specific execution semantics for different DSL instances. The need for a uniform and flexible mechanism to integrate OCL with arbitrary DSLs is evident.

4.2 Usage Scenarios

The previous section has already illustrated a number of usage scenarios for OCL in the context of domain-specific languages. In this section, I will provide a more structured view on this topic. This is meant as a precursor for the requirements analysis provided in Section 4.4.

The use of OCL can be classified in two orthogonal dimensions. Firstly, OCL execution semantics vary strongly between model space (meta layers M1 and above) and system space (M0). I introduced this phenomenon back in Section 2.1.3. Secondly, on each meta level OCL is equally useful as a constraint and definition language for precise modeling as well as a query language for model transformations. Recalling Section 2.3.4, the constraint language aspect could be further subdivided according to the various forms of constraints supported by OCL. Section 2.2.2 has shown examples for the model transformation facet.

	Constraint Language	**Query Language**
Model Space (M1–M3)	Evaluation of DSL wellformedness rules on models	Model-to-model transformations
System Space (M0)	Evaluation of system invariants, derived values, operations etc.	Data warehousing, data transformation

Table 4.1: *Two-dimensional classification of OCL usage scenarios*

Table 4.1 summarizes the ideas presented in this section by showing both dimensions of OCL usage and corresponding application scenarios. The following sections will further elaborate on this classification.

4.3 A Conceptual Framework

4.3.1 Overview

To guide my investigations and cleanly separate the different aspects of my work, I have devised a simple research framework. All sections in Chapters 5 and 6 will be structured according to this framework. In addition, this approach helps to limit the scope of this report. The elements of the framework are shown in Figure 4.3.

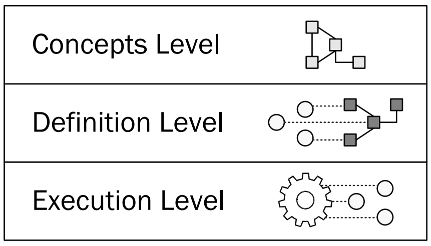


Figure 4.3: *Three levels of OCL integration*

The key observation is that integrating OCL with arbitrary domain-specific languages has to occur on three different levels. Below is a short, introductory description of each level. Subsequent subsections will provide a more thorough discussion. The descriptions use the notion of a *target language* which I define as the domain-specific modeling language that is to be integrated with OCL. Models in this language are supposed to be enriched by OCL expressions and are called *target language instances*.

The Concepts Level defines the metamodel of the target language whose elements are referenced by the elements of the OCL metamodel.

The Definition Level establishes the links between the instantiated OCL metamodel elements and a target language instance. These instances do not necessarily exist in the same model repository.

The Execution Level is where an OCL engine (i.e., an interpreter or compiler) walks the object graph of the instantiated OCL metamodel elements and applies its semantics to a concrete instantiation of the target language instance.

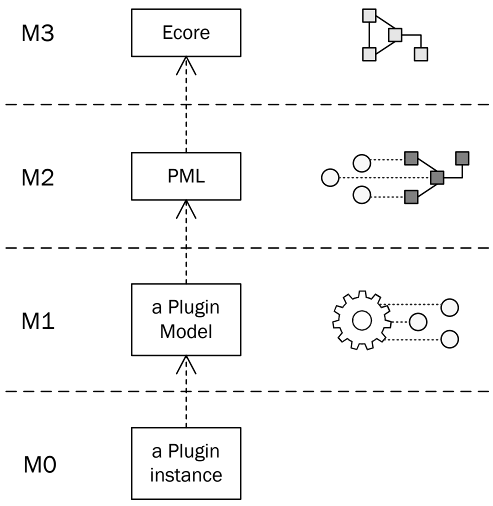


Figure 4.4: *OCL integration for execution in model space*

I have deliberately chosen a new naming scheme for these levels rather than resorting to the familiar terms employed in the meta hierarchy introduced in Section 2.1.3. This is because the three levels can be shifted along the traditional meta layers depending on the concrete usage scenario. Figures 4.4 and 4.5 exemplify this observation by showing the complete meta stack for PML and its integration into my conceptual framework.

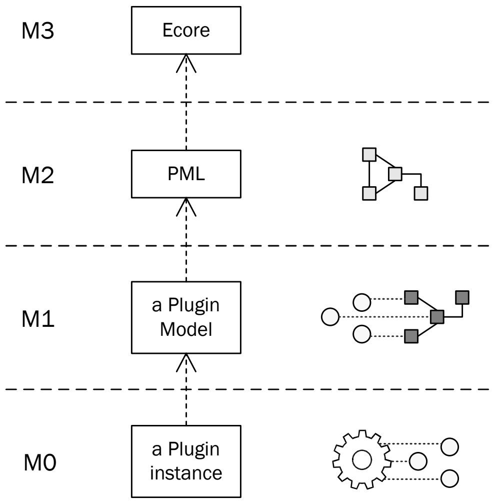


Figure 4.5: *OCL integration for execution in system space*

4.3.2 The Concepts Level

A domain-specific language that is supposed to be integrated with OCL needs to support a number of concepts. Logically, defining operation call expressions does not make sense without the notion of an Operation in the target language. However, the question arises whether it is necessary to support *all* concepts supported by OCL. In fact, it is highly unlikely that an arbitrary DSL will, to a large extent, be isomorphic to the UML metamodel which OCL has been designed for. It may well be that several target language concepts in conjunction provide the information required by a particular OCL abstract syntax element. Some aspects might even be missing, can be deduced from the context or have a constant value for all instances of the DSL metamodel. For instance, in Section 4.1, I interpreted the PML ExtensionPoint element as a multi-valued attribute, i.e. from the OCL point of view this element will always have an unlimited upper multiplicity bound.

The challenges of the Concepts Level are a direct result of the ontological classification problem introduced in Section 2.1.3. As I have already stated there, mapping ontological concepts of a DSL to the linguistic elements expected by OCL may often prove difficult, sometimes impossible.

Certainly, a DSL does not need to support all concepts provided by the UML to be OCL-conformant. The Concepts Level therefore needs to address the question how far a metamodel can be simplified while still offering a solid basis for OCL expressions. In the context of model transformations and meta-hierarchy integration researchers have suggested graph-based metamodels that essentially only define two concepts – nodes and lines [KvdB04, GFB05]. In theory, every model can be expressed as an instance of this simple metamodel. Yet, due to the limited expressiveness the resulting object graphs tend to become large and difficult to understand. The complex diagram in Figure 4.6 depicting an excerpt of the MOF metamodel illustrates this problem.

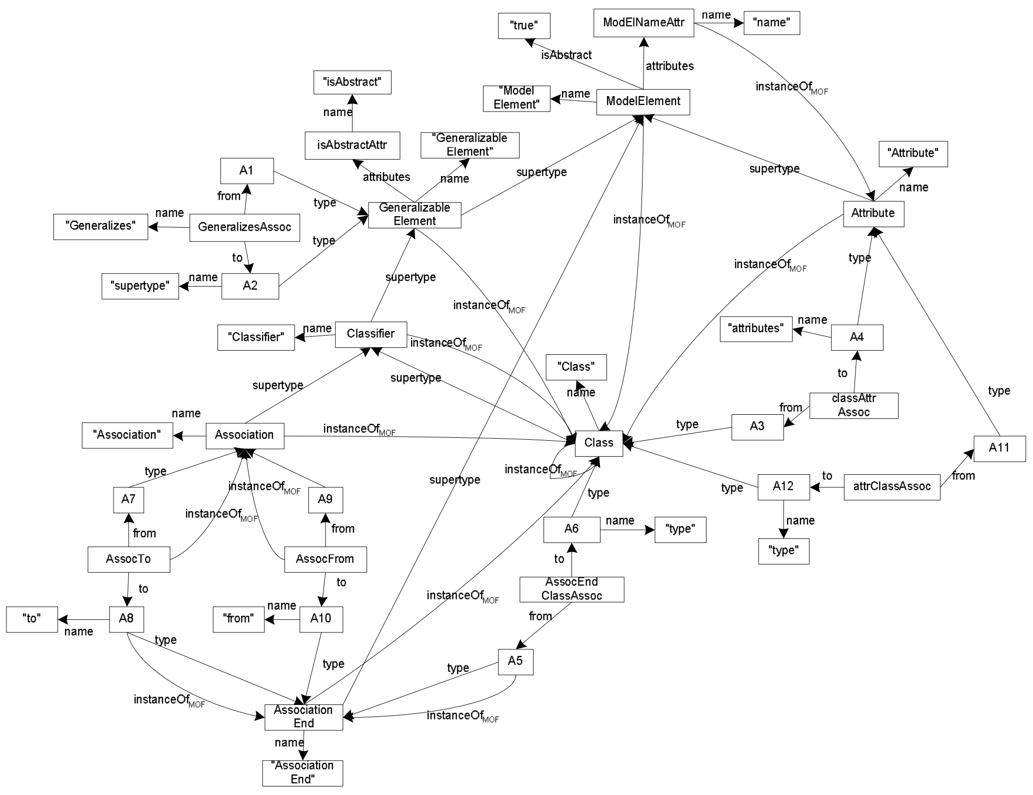


Figure 4.6: *Graph-based representation of the MOF model (taken from [KvdB04])*

Evidently, this representation lacks clarity and may not be well-suited for OCL constraints and queries. Furthermore, the entire static and dynamic semantics of the OCL metamodel would have to be redefined to fit the graph-based metamodel. Section 4.5 will propose the idea of a core Pivot Model as a solution.

4.3.3 The Definition Level

Once the OCL-relevant concepts of a target language have been identified, a mechanism has to be found that enables connecting the instances of both the OCL metamodel and the DSL's metamodel. Generally-speaking, this is one of several *metamodel composition* problems, which recently have started to receive more attention in the MDSD community [ES06, KPKP06, FBV06].

In the following, I will present selected metamodel composition techniques that seem adequate to solve the OCL-DSL integration problem. Many solutions have been proposed, but I will limit the discussion to those relevant in the context of this report. Beforehand, one more remark is necessary: in a heterogeneous modeling environment, different forms of model repositories are imaginable. The Netbeans Metadata Repository (MDR) [MDR] and the EMF infrastructure (presented in Section 3.2) are well-known examples. Thus, a model integration approach should ideally be suited for linking elements from arbitrary data sources.

Model Merge

Merging is one of the most obvious choices for composing related metamodels. Semantically equivalent elements are merged into a single one whereas concepts without a corresponding counterpart are simply copied into the resulting metamodel. Figure 4.7 shows a simplified example. Current research focuses on two aspects of the merge. Firstly, how can semantically similar elements be identified [FBV06], and secondly, what actual operation is used for the merge [PB03].

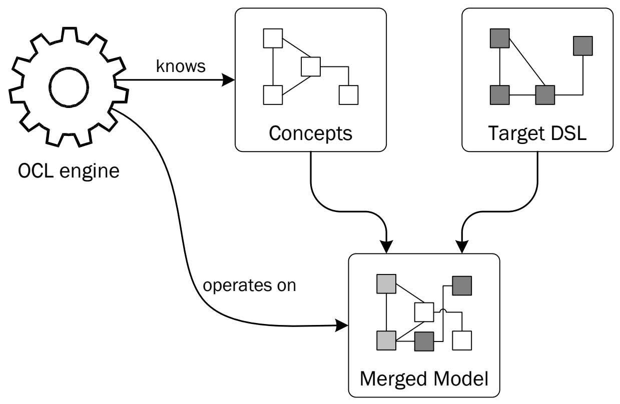


Figure 4.7: *Model composition using model merge*

To address the first issue, experience from schema integration in relational databases has proven to be useful [MZ98, DR06]. For instance, equal names or identifiers with a small Bayesian distance may be evidence of similarity. More advanced methods also analyze structural congruency and metamodel heuristics [FV07]. Finally, semi-automatic or manual methods requiring user interaction [FBJ$^+$05] as well as rule-based solutions [KPP06b] exist. A current trend is to visualize the correspondences between two models using a dedicated linking or weaving model which is itself an instance of an extensible weaving metamodel [SOE02, FBV06].

The second problem can be approached by many different means. The MOF Package Merge mechanism, which was introduced in Section 2.3.1, operates on the package level and can thus be understood as a "recursive unioning of model elements matched by name and metatype" [ES06]. It should be noted, though, that Package Merge merely implies the result of a merge, but does not denote it explicitly. More fine-grained operations, such as the Class Equivalence operator in MetaGME [LMB$^+$01], permit merging on the class level. A recent development in Feature-Oriented Programming (FOP) [BBGN01] and Product Line Engineering (PLE) [CN01] is the use of graph-rewriting systems (GRS) to facilitate model compositions [HL06]. However, this approach requires the metamodels to conform to the same meta-metalanguage. Other methods focus on role-based designs and define template slots that are filled with elements from another metamodel [ZL01, ES06]. Lastly, a very simple composition scheme is the inheritance relation in object-oriented modeling languages.

Although metamodel merge appears attractive due to its simple basic concepts and well-researched domain, it has some important drawbacks. To start with, merging elements that are not based on a one-to-one mapping is difficult. Furthermore, the merged elements may become "heavy-weight" and expose a bloated interface to clients like an OCL interpreter. Lastly, integrating metamodels from diverse model repositories remains an open question. Hence, I will not employ metamodel merge in my work.

Model transformation

An approach to model composition suggested by [Wen06b] is the use of *model transformations*. Suitable transformation rules may convert a target language instance into the concepts representation required by the client. Figure 4.8 illustrates this idea. For instance, Abouzahra et al. discuss the transformation of arbitrary DSL models into corresponding UML representations [ABFJ05]. Since many UML-based OCL engines are readily available, this may be a promising solution for easy OCL integration. Of course, OCL expressions that have originally been defined for the target language instance need to be transformed as well, which may pose difficulties.

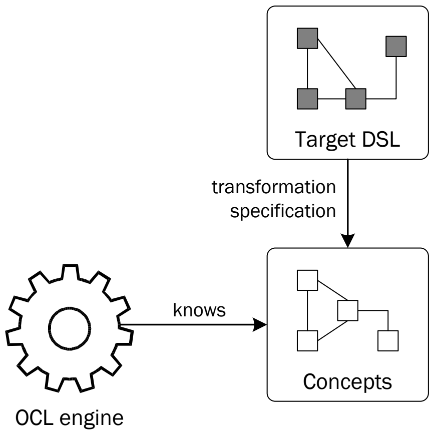


Figure 4.8: *Model composition using model transformation*

Time constraints did not allow me to further follow this interesting idea. Its biggest benefit certainly is the use of a declarative transformation language like QVT or ATL to specify the composition algorithm. Still, the problem of heterogeneous model repositories remains unsolved.

Model interfacing

As outlined in Section 2.1.2, current research in the area of domain-specific languages aims at describing a system with a variety of DSLs whose models are then composed to a complete system specification. For instance, a model capturing requirements may be connected with a GUI specification and a DSL instance describing the logical core. An answer to this problem is *model interfacing* which essentially means the insertion of interface model elements that “glue” together the different model parts [ES06]. The basic idea is captured in Figure 4.9. Foundational work in this area is presented in [BSH99].

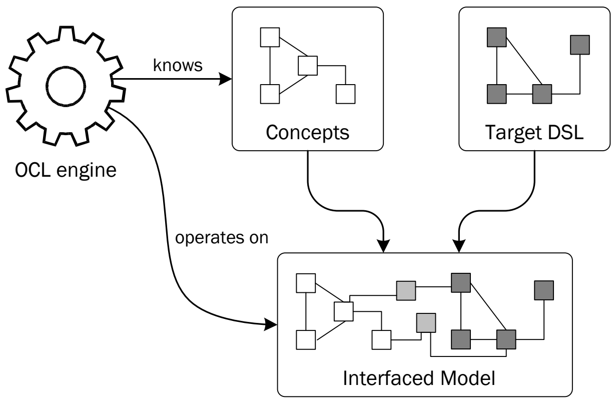


Figure 4.9: *Model composition using model interfacing*

However, to the best of my knowledge, model interfacing is not a mature engineering methodology yet. Currently, the integration process is often deferred until code for the target platform has been generated [WK06]. Rather than composing on the model layer, code fragments generated from individual models are integrated with appropriate “glue code”. Obviously, this approach sacrifices a higher level of abstraction for practicability and feasibility. Indeed, the model adaptation mechanism I have eventually used in this project (see below) follows a similar pattern. The main difference is that model interfacing emphasizes the role of links between models of equal priority whereas model adaptation employs an adapter layer that adapts one of two domain-specific languages to the other.

Model adaptation

Model adaptation is a novel term for model composition that has not been described extensively in the literature so far. I use it to denote the rather pragmatic, object-oriented approach that I have realized for this report. Its core idea is to use a modern code generation facility like EMF to create an interface layer in an object-oriented programming language such as Java. These interfaces can then be implemented as adapters [GHJV95] to an arbitrary domain-specific target language. Thus, an OCL processor can rely on the interface layer and does not need more knowledge about the DSL behind it. A sketch of this idea is shown in Figure 4.10.

The benefits of this solution are its simplicity, flexibility and extensibility. Adapting the structure of a target language using intuitive, imperative code provides practically limitless possibilities for an experienced developer. Integrating different model repositories does not pose any problems either because implementation details can be hidden in the adapters. Finally, for straightforward one-to-one mappings large parts of the implementation skeleton can be generated automatically based on given correspondences between the interface layer and the target language.

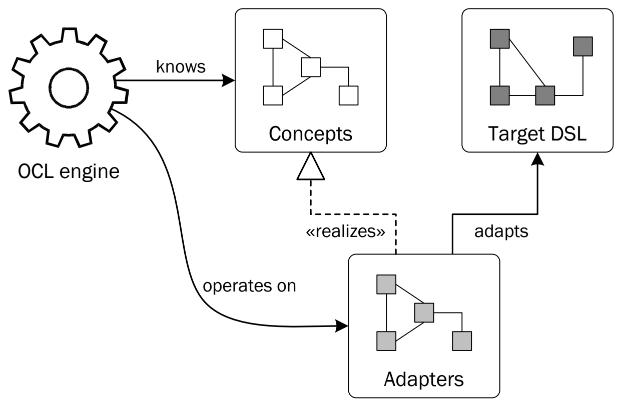


Figure 4.10: *Model composition using model adaptation*

Of course, when compared to a declarative, model-based mapping definition this approach also reveals some significant disadvantages. In particular, it sacrifices platform independence and generality. Furthermore, implementing the DSL integration in imperative code is more error-prone and has fewer reuse potential. Essentially, all pro and contra considerations between traditional software engineering and model-driven development apply. Yet, I believe that carefully designed, abstract layers of reusable integration code as well as code generation techniques can achieve an acceptable tradeoff between flexibility and universality.

4.3.4 The Execution Level

The Execution level is actually outside the scope of this report. However, to provide a sound basis for future work, I will include a brief discussion of this level when analyzing related work (Chapter 5) and presenting the results of my project (Chapter 6).

The major challenge on the Execution Level is the integration of the OCL Standard Library with the instance of the target language model. The predefined data types and operations of the Standard Library have to be available to the OCL engine. However, a simple mapping to a programming language like Java (as described in [WK03]) may not suffice due to special execution semantics of the domain-specific type system. For instance, an OCL Real instance does not necessarily map to a `java.lang.Float`. Similarly, collection and tuple types may have domain-specific representations. Passing an OCL type to an operation of a domain object thus requires a conversion.

In general, to support arbitrary DSLs, all types defined in the OCL Standard Library need to be convertible to a domain-specific representation. Likewise, all domain-specific types have to be reconverted when they are returned as the result of an operation or property call. Clearly, a flexible and generic mechanism to facilitate these conversions is required. Ideally, the domain-specific OCL representations themselves could capture some of this knowledge. In Section 6.1.3, I will suggest an adapter-based solution addressing these requirements.

Another challenge to be met on the Execution Level stems from the system instantiation problem introduced back in Section 2.1.3. If OCL is used in the context of object transformations, it must be possible to create new instances of domain objects in the target system. Again, this is highly DSL-specific. In the domain of Java programs, objects are instantiated using the `new` operator. The semantics for creating rows in a relational database, however, are usually much different.

4.4 Requirements Analysis

Now that I have analyzed the problem domain and established a conceptual framework, I can devise a concise list of requirements as a guide for the remainder of this report. The intent of these requirements and their association with the different levels of my research framework directly follows from the previous sections, so I will give no further explanations.

In summary, the goals of this report are:

1. capturing all concepts that may be supported by domain-specific languages in order to be integrated with the Object Constraint Language
2. designing and implementing a flexible and extensible metamodel composition mechanism to map arbitrary DSLs onto the concepts defined for the first requirement
3. providing means to enable this integration for heterogeneous model repositories
4. proposing a flexible and extensible mechanism for integrating the OCL Standard Library with system objects from arbitrary domains

4.5 The Idea of a Pivot Model

Finally, I can properly put the central topic of this report in context. The first requirement listed in the last section already points towards a central, common metamodel that contains all the concepts (and only those) that are referenced by the OCL metamodel. In [Wen06b], the term *pivot model* is introduced to describe such a metamodel. Given an integration scheme that satisfies the second requirement from the last section, a pivot model could serve as a central exchange format for models and metamodels in an MDSD environment. Precise modeling with OCL and model transformations with an OCL-based language become possible on all metalevels and for all domain-specific languages that have a mapping to and from the pivot representation. The next chapter will investigate to what extent related works have already realized this level of domain independence.

5 Related Work

This chapter thoroughly analyzes the respective strengths and weaknesses of three different projects that, to a certain extent, already support the integration of OCL with multiple metamodels. The chosen projects comprise the Dresden OCL2 Toolkit, the Kent OCL Library and the Epsilon Platform. To the best of my knowledge, they represent the only published work targeting similar goals as this report. The evaluation presented in this chapter is organized along the structure of the conceptual framework introduced in Section 4.3.

5.1 The Dresden OCL2 Toolkit

5.1.1 Overview

The Dresden OCL Toolkit [TUD] is a collection of tools and libraries that has been under development at Dresden University of Technology since 1999. The current 2.0 release features a metamodel-based architecture [Ock03, LO04], which is built around the Netbeans Metadata Repository [MDR], and a parser generated from an L-attributed grammar of OCL 2.0 [Kon03, DHK05]. A comprehensive overview of the toolkit architecture and the tools developed around it can be found in [Wen06b].

A remarkable feature of the current toolkit architecture is the implementation of a metamodel integration mechanism. This stems from the fact that at the time of creation, the MOF and UML metamodels had not yet been aligned and, thus, did not share a common core. In great similarity to the ideas presented in Chapter 4, the toolkit accesses MOF and UML models via the interfaces defined in a common metamodel. In the following sections, I will analyze the toolkit's implementation in respect to the conceptual framework I have introduced in Section 4.3.

Note that when the current version of the toolkit was developed, the OCL 2.0 specification was still in its finalization stage. As a result, the most recent version of the specification [OMG06c] deviates in some details from the metamodel implemented in the toolkit. I will point out some differences in this chapter, but will leave other points for discussion when introducing my own solution in Chapter 6.

5.1.2 The Concepts Level

To provide a common interface for both MOF and UML models, the Dresden OCL2 Toolkit defines a *CommonModel* as part of a Common-OCL package which also contains the OCL Expressions and Types packages. The CommonModel represents an abstraction of the UML and MOF metamodels that contains all the concepts referenced by the OCL metamodel. Obviously, this corresponds to the Pivot Model idea presented in the previous chapter.

However, the scope of the CommonModel is significantly limited compared to the goals of this report. Whereas the Pivot Model is supposed to provide a common base for an arbitrary number of domain-specific languages, the CommonModel merely serves as a bridge between the structurally different metamodels of UML 1.5 [OMGb] and MOF 1.4 [OMG02]. This is reflected by its design which, to a large extent, corresponds to the UML metamodel, leaving out only a few metaclasses that are not relevant for the binding to OCL. The biggest disadvantage of this approach is that structural deficiencies and conceptual flaws of the UML 1.5 metamodel also exist in the CommonModel.

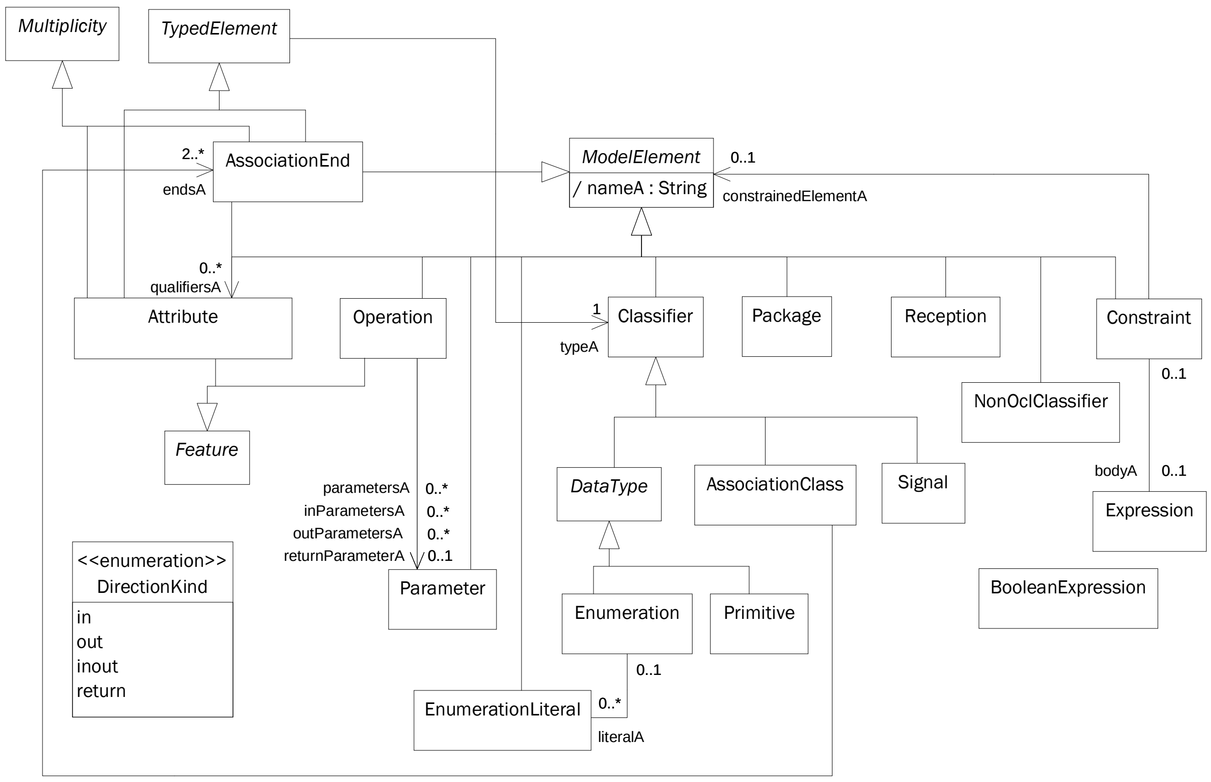


Figure 5.1: *The CommonModel of the Dresden OCL2 Toolkit*

Figure 5.1 shows the inheritance hierarchy of the metaclasses in the CommonModel (note that the suffix A in the derived attribute nameA stands for "abstract" and is a special naming convention in the Toolkit to highlight the difference between the CommonModel and the integrated metamodels of UML and MOF). Following is a list of observations that serve as a guide for the design of my own Pivot Model.

To begin with, the CommonModel contains a number of **UML-specific metaclasses** such as Signal, AssociationClass and Reception. Clearly, including these classes is undesirable when aiming for generality in regard to arbitrary domain-specific languages. This does not necessarily mean a loss in expressiveness because concepts of behavioral modeling languages may be mapped to more basic Pivot Model classes. Section 4.1 argued that, given a proper interpretation, different semantics may be achieved with OCL expressions.

Another issue worth highlighting is the metaclass NonOclClassifier that is mixed into the MOF- and UML-specific implementations of the CommonModel Classifier concept. It defines an operation `toOclType` that maps predefined MOF and UML data types to the corresponding OCL types. Unfortunately, this design introduces **dependencies from the CommonModel to the OCL metamodel**. In the current toolkit architecture this does not pose major problems, because the CommonModel is part of the Common-OCL package anyways. In this project, however, I will strive to achieve a layered architecture, where the Pivot Model is free of external dependencies. This paves the way for the integration with different constraint and query languages and also allows for a cleaner system architecture (see Section 6.2).

A problem that stems from the legacy of the outdated UML metamodel is that **Operation does not extend TypedElement**. In fact, the UML 1.4.2 specification [OMG04] does not even define the abstraction TypedElement. In contrast, most modern metamodels for class structures define a type for operations which usually corresponds to the operation's return type. Hence, this notion should be included in the Pivot Model as well. A beneficial side effect is the simpler type evaluation of OCL operation call expression.

Finally, a useful pattern in the design of the CommonModel is the **use of multiple inheritance** to mix in properties shared among disjunct sets of meta classes. For instance, Parameter is a TypedElement just like Attribute and AssociationEnd, but it does not share the inheritance from Feature, which, on the other hand, is extended by Operation. Seemingly, metamodels often contain those overlapping inheritance structures. Sometimes, it may be possible to construct a hierarchy using only a single line of inheritance but I do not see the benefit of doing so. After all, this usually only serves to provide implementation reuse when metamodels are mapped to a concrete representation (e.g., Java classes). On the conceptual level, using multiple inheritance to define semantically distinct features of metaclasses should be preferred.

5.1.3 The Definition Level

Model composition

The current Dresden OCL toolkit implementation uses inheritance for model composition. Figure 5.2 exemplifies this for the inheritance relationships between the CommonModel elements Classifier and Operation and their corresponding counterparts in the UML metamodel.

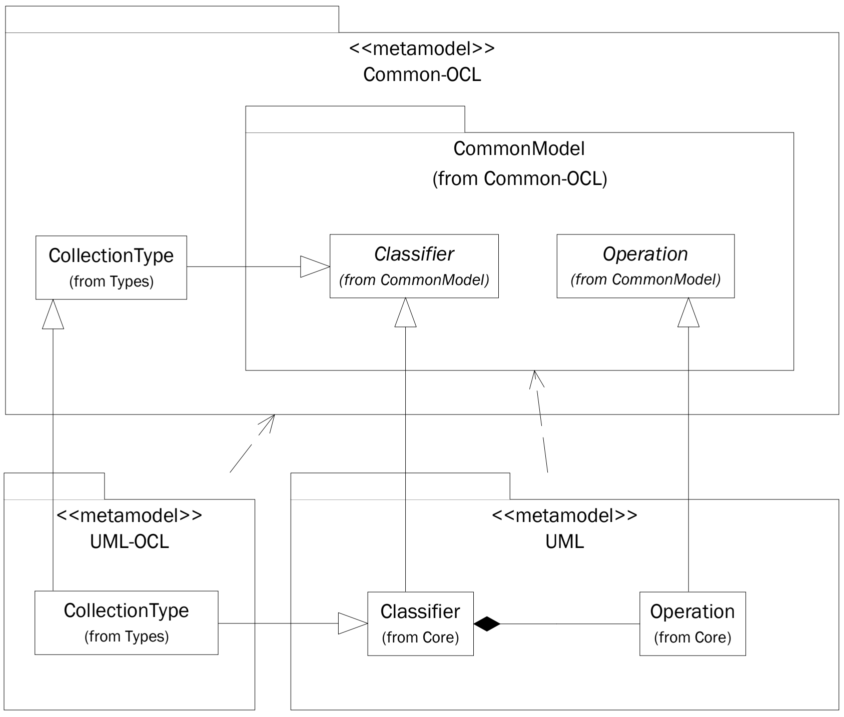


Figure 5.2: *Inheritance as metamodel composition technique in the Dresden OCL2 Toolkit*

One major drawback of this approach is that an additional UML-OCL package is necessary to define the OCL elements that inherit from UML metamodel classes. Every new binding of a DSL with OCL requires such a metamodel-specific package for the OCL elements. The reason is that associations on the CommonModel level cannot be reused within the DSL-specific metamodel. Note that the CommonModel elements Classifier and Operation are not connected via an association. In essence, this is an example of the *Class Adapter* design pattern [GHJV95] and the resulting disadvantages of mixing interface and implementation inheritance. Although the interfaces for the specific OCL elements can be generated automatically, about 800 lines of implementation code are duplicated in the Expressions and Types packages of the UML-OCL and MOF-OCL bindings, respectively.

Moreover, this approach alone does not suffice to integrate languages with a structure that is different from the UML. Consequently, the MOF-OCL package provides an additional set of adapters that realize the CommonModel semantics using a combination of the Class and Object Adapter pattern. Figure 5.3 illustrates the principle for the CommonModel Datatype element which is mapped to the MOF Datatype metaclass. The adapter class AdDataType additionally needs to extend MOF::Class because Datatypes in MOF do not possess Operations.

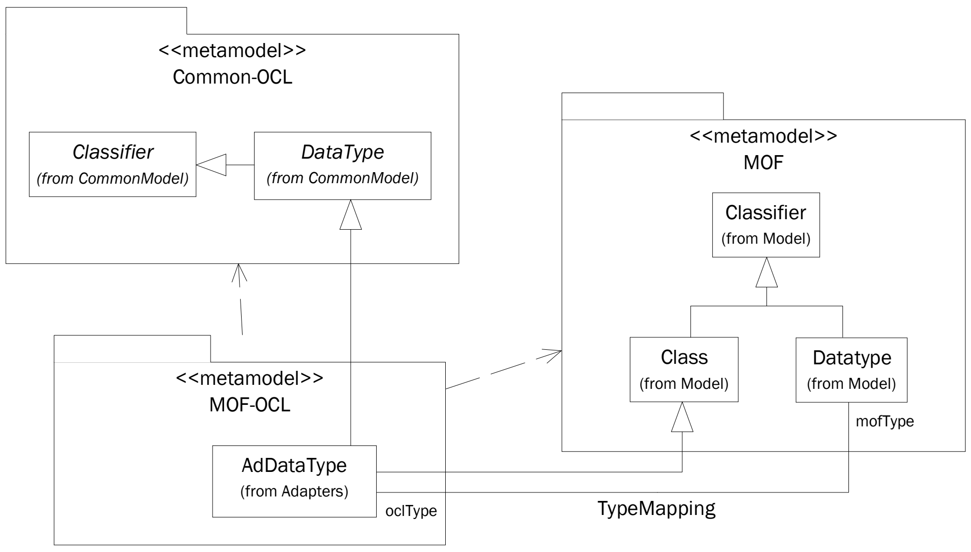


Figure 5.3: *Metamodel adaptation in the Dresden OCL2 Toolkit*

Accessing different model repositories

The original design of the Dresden OCL2 Toolkit did not allow evaluating OCL constraints over models stored in other repositories. The metamodels of OCL and the target language were integrated in the Netbeans MDR and their instance models were assumed to be loaded into the same repository. These shortcomings were addressed when the toolkit was integrated [Stölzel05, SZG06] with the Fujaba Tool Suite [Fuj]. Figure 5.4 gives an overview of the mechanism developed for this purpose.

The core idea of the new design is to represent model elements in the custom repository with proxy objects stored in the Netbeans MDR. Instances of the OCL metamodel elements then reference these proxies. A central Singleton class called `ModelFacade` maps between the different representations using the MOF identifier of the proxy in the MDR as a key. Navigating the model and accessing properties of the model elements is solely done via the `ModelFacade`. To this end, it provides getter methods representing the union of all attributes and associations of each metaclass defined in the UML metamodel.

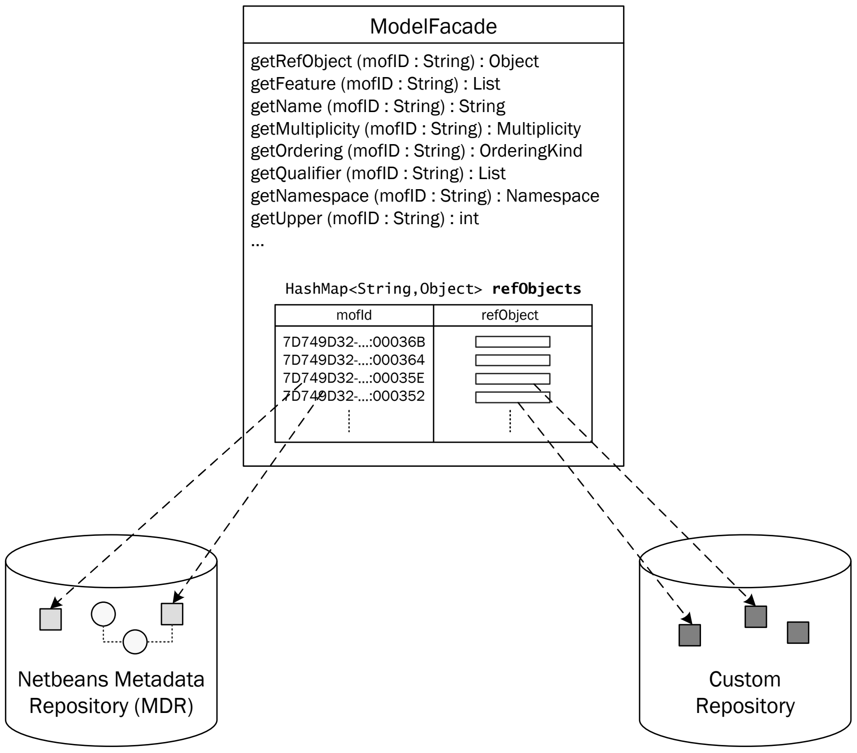

ModelFacade
getRefObject (mofID : String) : Object
getFeature (mofID : String) : List
getName (mofID : String) : String
getMultiplicity (mofID : String) : Multiplicity
getOrdering (mofID : String) : OrderingKind
getQualifier (mofID : String) : List
getNamespace (mofID : String) : Namespace
getUpper (mofID : String) : int
...
HashMap<String,Object> refObjects
mofId
refObject
7D749D32-...:00036B
7D749D32-...:000364
7D749D32-...:00035E
7D749D32-...:000352
Netbeans Metadata Repository (MDR)
Custom Repository

Figure 5.4: *The ModelFacade design of the Dresden OCL2 Toolkit*

Evidently, this design yields a number of disadvantages. Most importantly, accessing model properties via a central class crosscuts the implementation of every meta element and thus results in very monolithic and inflexible code that is difficult to maintain. Secondly, placing all accessor methods of a metamodel in a single class violates the object-oriented principles of encapsulation and separation of concerns. Keying the custom repository elements via the MOF identifier poses additional problems for compile time type checking. Finally, the interface provided by the `ModelFacade` solely targets UML models and cannot easily be readjusted for arbitrary domain-specific languages.

5.1.4 The Execution Level

The Dresden OCL2 Toolkit currently features a Java code generator for wellformedness rules over MOF metamodels [Ock03] and constraints defined on UML model elements [Bra06].[1] The generated code uses an extended version of the OCL Standard Library implementation presented in [Fin99]. Its class hierachy is shown in Figures 5.5 and 5.6. Note that the second diagram represents the meta programming layer of the OCL Standard Library which is required for class-scope operations and runtime type checking (e.g., in `oclIsKindOf`).

Unfortunately, the current Standard Library implementation fails to fulfill the requirement of enabling domain-specific representations for all OCL predefined types (see Section 4.3.4). Adapting the instances of domain-specific languages is only possible for objects, types, enumeration literals and enumeration types. Figures 5.5 and 5.6 show the corresponding implementations for MOF and UML with a yellow fill color (note that the current version of the toolkit does not yet provide an implementation of `OclEnumLiteral` and `OclEnumType` for UML). In contrast,

[1] Additional generators allow to transform integrity rules over data models into corresponding SQL constraints [Hei05] or declarative code [Hei06].

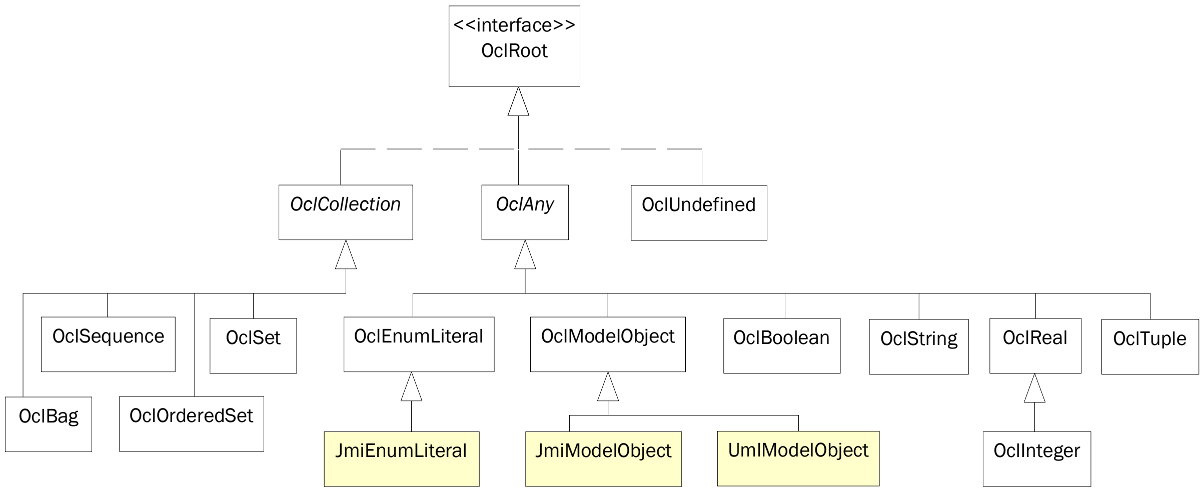


Figure 5.5: *The Standard Library implementation of the Dresden OCL2 Toolkit (Objects)*

the OCL primitive types as well as the collection types have fixed implementations backed by suitable Java types.

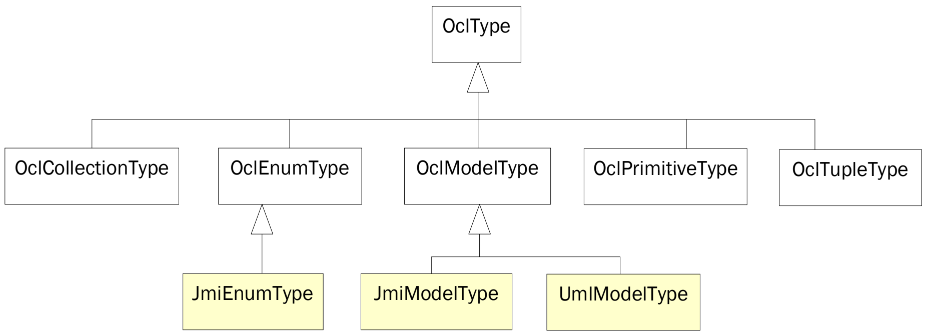


Figure 5.6: *The Standard Library implementation of the Dresden OCL2 Toolkit (Types)*

Converting objects between the OCL representation and the target language instance is accomplished through an interface `OclFactory` and its methods `getOclRepresentationFor` and `reconvert`. Metamodel-specific implementations of this interface (currently `JmiOclFactory` and `UmlOclFactory`) are realized as Singletons which are accessed by the generated code. Figure 5.7 exemplifies the core principle showing the conversion to and from JMI representations, respectively.

Once again, the current design relies on a central entity doing all the work which hinders encapsulation and worsens maintainability. Indeed, the complex control flow in the `reconvert` method in `JmiOclFactory` causes a very high *McCabe Cyclomatic Complexity* [MW94] of 59.[2]

Another issue adding complexity is that both conversion methods in `OclFactory` require an explicit target type to be specified. The motivation is to denote the expected OCL or domain-specific type, respectively, since it cannot always be deduced from the other representation. However, a look at the implementation reveals that this parameter is only effectively used in two circumstances:

[2] The McCabe Cyclomatic Complexity measures the number of linearly independent paths through a function, module or program and is a good indicator of its complexity.

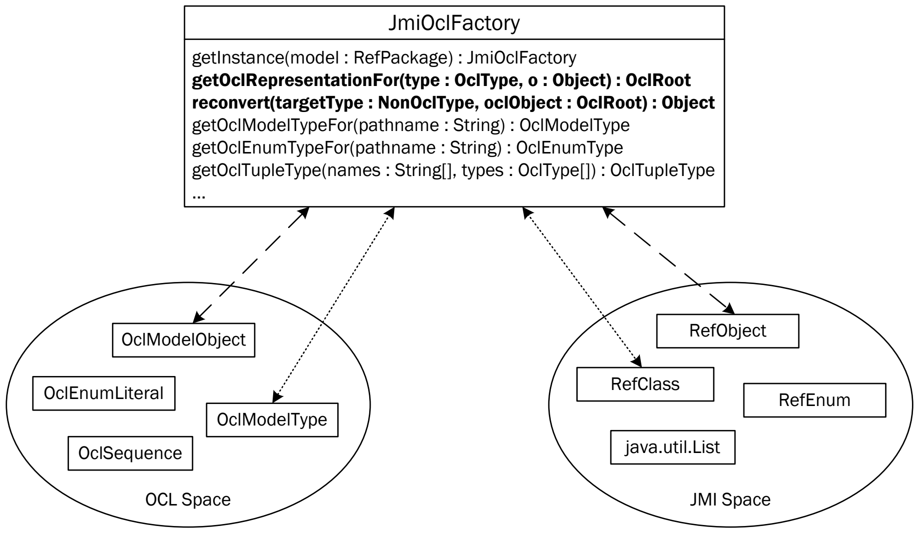

JmiOclFactory
getInstance(model : RefPackage) : JmiOclFactory
getOclRepresentationFor(type : OclType, o : Object) : OclRoot
reconvert(targetType : NonOclType, oclObject : OclRoot) : Object
getOclModelTypeFor(pathname : String) : OclModelType
getOclEnumTypeFor(pathname : String) : OclEnumType
getOclTupleType(names : String[], types : OclType[]) : OclTupleType
...
OclModelObject
OclEnumLiteral
OclModelType
OclSequence
OCL Space
RefObject
RefClass
RefEnum
java.util.List
JMI Space

Figure 5.7: *Converting between OCL and JMI representation in the Dresden OCL2 Toolkit*

- a domain-specific collection instance (e.g., `java.util.Collection`) needs to be converted to the correct OCL type (`OclSequence`, `OclSet`)
- an OCL primitive type (e.g. `Integer`) needs to be reconverted to the domain-specific type (`java.lang.Integer`, `java.lang.Long`, etc.)

I argue that both cases can be approached differently. To convert collection instances, dedicated methods for each of the OCL collection types can be provided. It is unlikely that the number of collection types will increase dramatically in a future OCL specification, so this should not result in serious maintenance issues. Interestingly, my suggestion resembles the approach of the original Standard Library implementation in [Fin99]. The other problem of reconverting primitive types to their domain-specific representation is minimal in the context of domain-specific languages because the set of possible target language types will usually be limited. Modern programming language features such as automatic type conversion and automatic boxing and unboxing of primitive types should eliminate the need for an explicitly specified target type. After all, the current implementation also relies on these language capabilities to reconvert OCL primitive types into their corresponding primitive Java types (`boolean`, `int`, etc.).

An additional deficiency of the current Standard Library design emerges when considering its use in an (envisaged) OCL interpreter. Figures 5.8 and 5.9 highlight the problem by showing selected operations of the Standard Library classes. Of particular importance are the `getFeature` methods in `OclModelObject` and `OclModelType` that allow retrieving property values and invoking operations on domain-specific objects and types.

Now, the key observation here is that predefined OCL operations (such as the iterator operations in `OclCollection` or the `oclAsType` operation in `OclAny`) cannot be called reflectively because the necessary `getFeature` method is only defined for `OclModelObject`. This applies similarly to operations in `OclType` (e.g., `allInstances`). This does not pose any problems for code generation because the method call is simply written to the generated Java file. However, an interpreter must be able to call all operations reflectively, even those defined in the OCL Standard Library.

Unfortunately, simply refactoring `getFeature` into the `OclRoot` interface does not suffice either. Consider that `getFeature(type,name,parameters)` takes `OclRoot` objects as parameters (wrapped into `OclParameter` instances). Since `OclType` does not extend `OclRoot`,

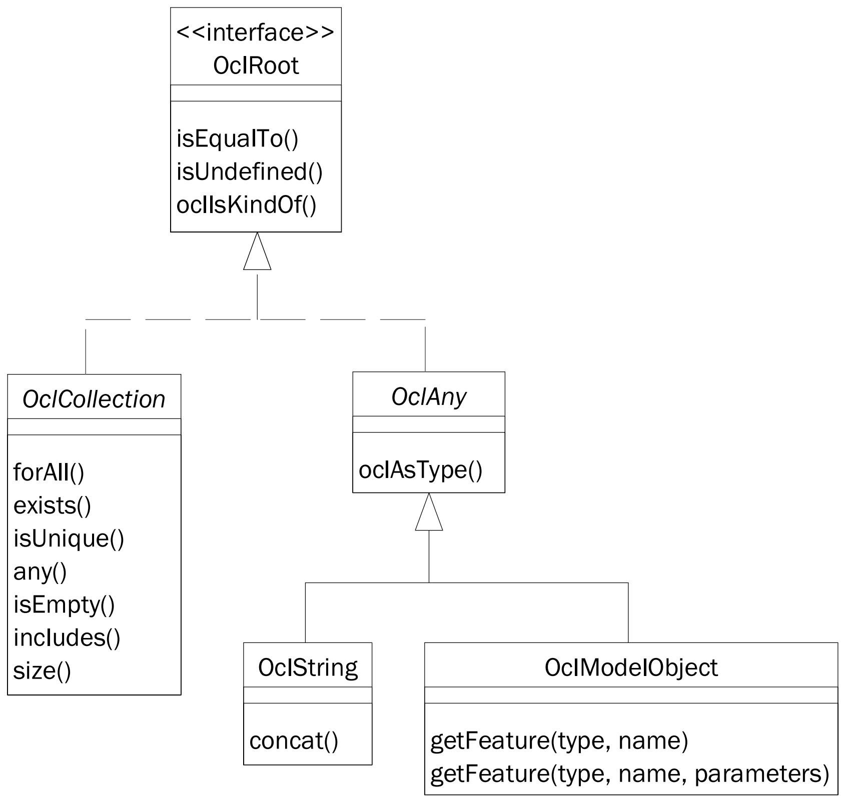


Figure 5.8: *Excerpt of Standard Library operations in the Dresden OCL2 Toolkit (Objects)*

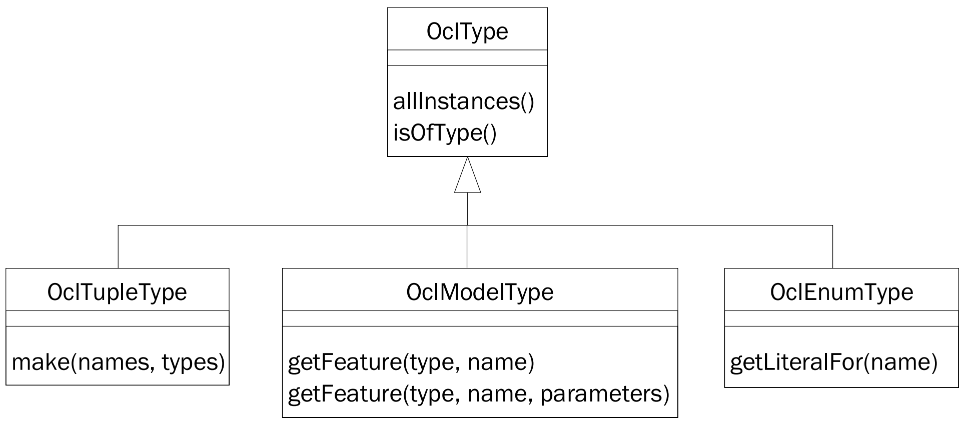


Figure 5.9: *Excerpt of Standard Library operations in the Dresden OCL2 Toolkit (Types)*

operations with type arguments (e.g., `oclAsType` or `oclIsKindOf`) cannot be called reflectively.

Finally, the current design requires a specialized code generator for each repository technology and metamodel. This stems from the fact that the concrete `OclFactory` implementation as well as the model format are directly written into the resulting Java file. Listing 5.1 shows an example created by the JMI code generator. Naturally, an abstraction for different model repositories is desirable, particularly with regard to an OCL interpreter.

```java
public class OclEvalBooleanLiteralExp {
  private RefPackage model;

  public OclEvalBooleanLiteralExp(RefPackage model) {
    this.model = model;
  }

  // invariant BooleanLiteralExpWFR2 : true
  public boolean evaluateBooleanLiteralExpWFR2() {
    final JmiOclFactory tudOcl20Fact0 = JmiOclFactory.getInstance(model);
    final OclBoolean tudOcl20Exp0 = OclBoolean.TRUE;
    return tudOcl20Exp0.isTrue();
  }
}
```

Listing 5.1: *Generated code with dependencies to the model repository technology*

In summary, the current Standard Library implementation in the Dresden OCL2 Toolkit does not satisfy the requirements for an integration with arbitrary domain-specific languages. Converting objects between OCL and domain space is cumbersome. Furthermore, it lacks the reflective capabilities and abstraction required by an OCL interpreter. In Section 6.2, I will suggest an alternative design that addresses these issues.

5.2 Kent OCL

5.2.1 Overview

The Kent Object Constraint Language Library [WWW] has been developed under the Kent Modeling Framework (KMF) project [KMF] at the University of Kent at Canterbury. It features an OCL parser, analyzer, evaluator and code generator [ALP03, AP04]. Most notably, it defines a *Bridge* metamodel as an abstraction over a number of different metamodels and provides implementations for Java, KMF and EMF. Thus, it follows a similar approach to the one presented in this report. The research in this area appears to be discontinued, though. Recent publications instead focus on detaching the OCL Standard Library from the language definition [AHMM06]. The following subsections will provide an analysis of the Kent OCL library within the bounds of the research framework presented in Section 4.3.

5.2.2 The Concepts Level

The structure of the Bridge metamodel is shown in Figure 5.10. Apparently, it is a heavily simplified version of the UML metamodel and barely resembles the Dresden OCL Toolkit CommonModel presented in Section 5.1.2. Naturally, the question arises whether such a reduced metamodel offers the amount of expressiveness required by an OCL engine. I argue that it does not, but its analysis was useful in revealing many important design considerations that I have incorporated in the Pivot Model.

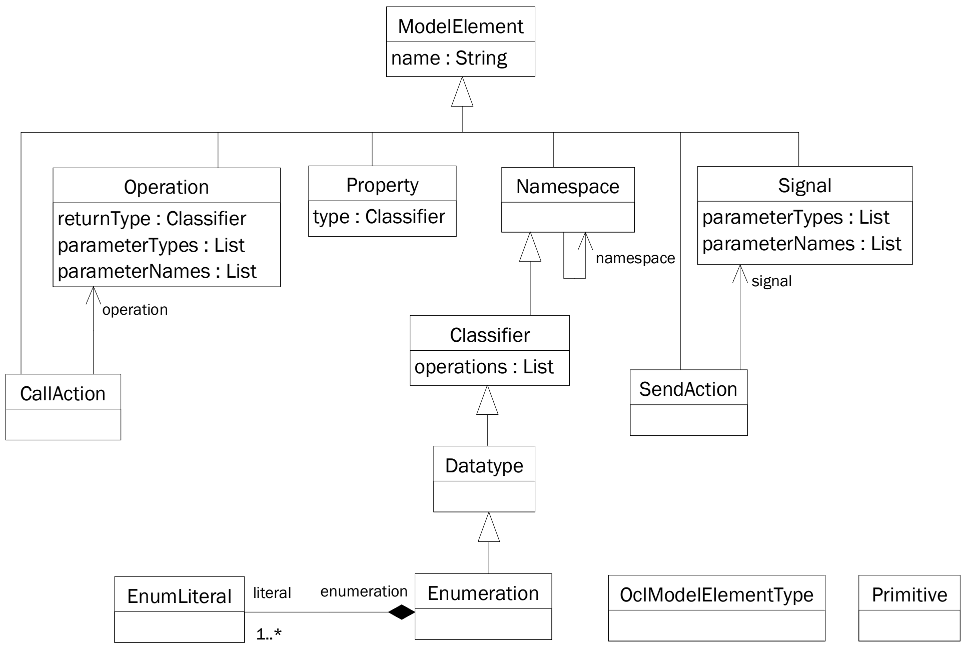


Figure 5.10: *The Bridge metamodel of the Kent OCL Library*

To begin with, the Bridge model does not support the notions of multiplicities and static features. Yet, the first concept is vital to support multi-valued model properties (and the implicit `collect` iterator operation) whereas the second is required for class-level property and operation calls. More of a cosmetic problem than an actual deficiency is the lack of an abstract metaclass TypedElement, although it would provide a better abstraction for the returnType attribute in Operation and the type attribute in Property.

Leaving out a Parameter metaclass is controversial, too. Certainly, it is not required to lookup Operations contained by Classifiers because only the types of the arguments count. Even so, a dedicated Parameter meta element allows for the definition of different parameter kinds (in, out, inout) which is explicitly supported by OCL.

An interesting observation from the Bridge model is that the Datatype metaclass is not actually required. Datatypes differ semantically from other Classifiers in that they are "identified only by their value" [OMG05d, p. 73]. For an OCL engine, this difference is irrelevant. Indeed, the corresponding interface in the Kent OCL implementation is empty and can be omitted. I will come back to this idea in Section 6.1.1.

In Figure 5.10, the two new metaclasses OclModelElementType (which represents user-defined types) and Primitive (which is the renamed UML PrimitiveType) have no visible connection to the other Bridge classes. This originates in an attempt to provide more consistency between the OCL metamodel and the OCL Standard Library. Rather than directly inheriting from Classifier or Datatype, primitive types and user-defined types derive from the OCL AnyType. Figure 5.11 displays the effects of this approach on the OCL Types package.

Compared with Figure 2.17 from Section 2.3.5, the structure of the Types package is significantly different. Main changes are the addition of the OclModelElementType and an altered inheritance hierarchy where AnyType (named OclAnyType here) becomes the superclass of non-

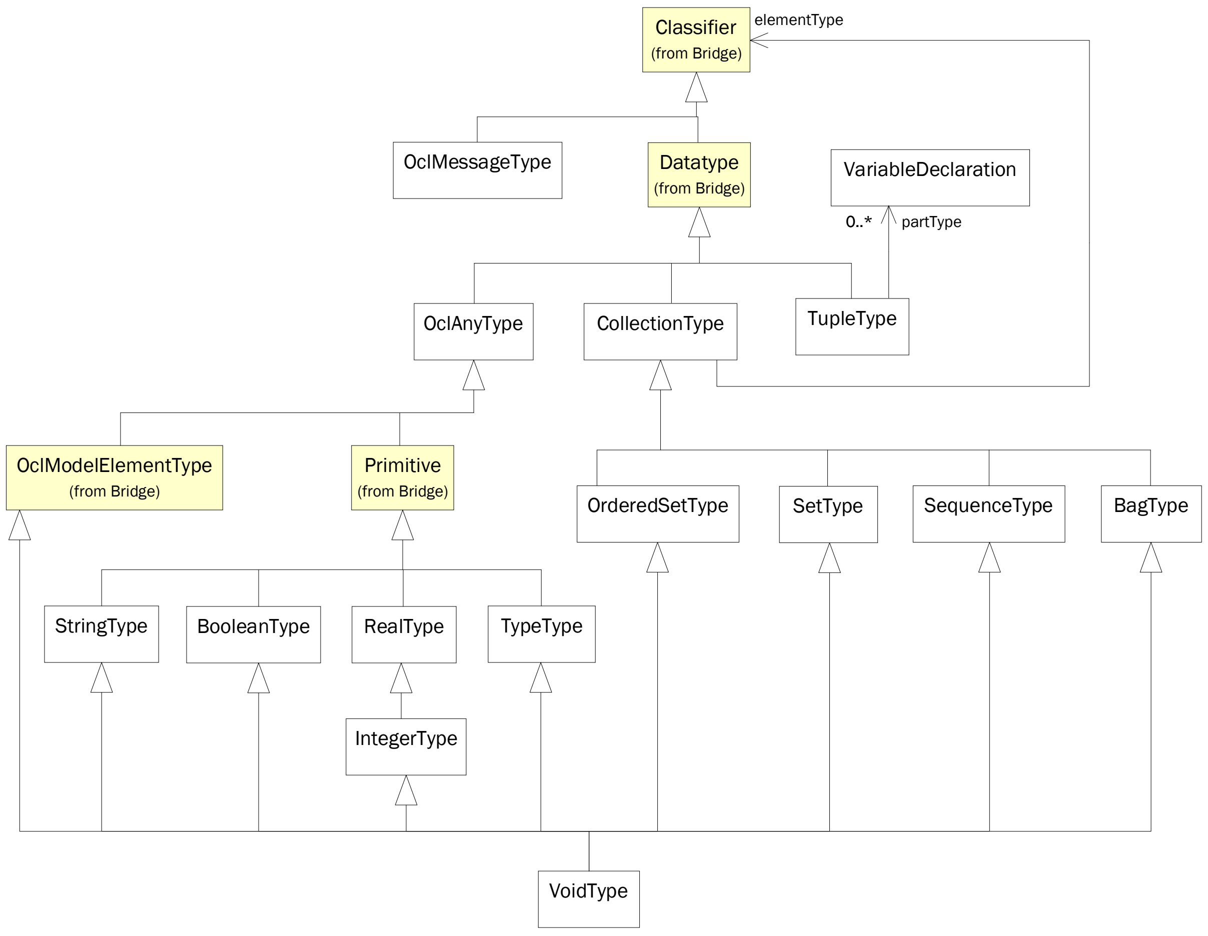


Figure 5.11: *The adjusted OCL Types package of the Kent OCL Library*

collection types, and VoidType descends from all other types. Essentially, the OCL metamodel (M2) has been aligned with the OCL Standard Library (M1).

However, this approach suffers from a fundamental conceptual flaw. In the MOF metamodeling architecture, inheritance relations are not propagated to a lower metalevel. This is easy to see when considering the primitive type Boolean which is an instance of the meta element Primitive. Even though Primitive extends Classifier on the M2 level, there is no corresponding inheritance relationship between Boolean and an arbitrary Classifier instance defined on M1. In particular, this does not yield the intended result of making Boolean a subtype of OclAny.

The only way to model cross-metalayer dependencies is the UML concept of *powertypes*. A powertype is a "metaclass whose instances are subclasses of a given class" [RJB04]. Consequently, modeling Primitive (M2) as a powertype of OclAny (M1) symbolizes that all instances of Primitive derive from OclAny. An inheritance relation between Primitive and OclAnyType on the Metamodel layer is not required. In fact, AnyType really only adds a semantic difference to the Classifier metaclass. It does not define any additional features, so its only benefit is that the single instance OclAny can be differentiated from "normal" Classifier instances in a meta repository via an `instanceof` check.

A noteworthy feature of the Kent OCL Library is the existence of an explicit model for context definitions (Figure 5.12) which does not rely on UML stereotypes to denote the various types of constraints supported by OCL. This paves the way for supporting constraints over arbitrary modeling languages. Unfortunately, in its disregard for the top-level concept ExpressionInOcl, the Kent OCL solution deviates strongly from the OCL specification [OMG06c, p. 159]. Moreover, it does not readily map to the UML/MOF 2.0 model of constraints which severely hinders an alignment with repositories realizing the new standards.

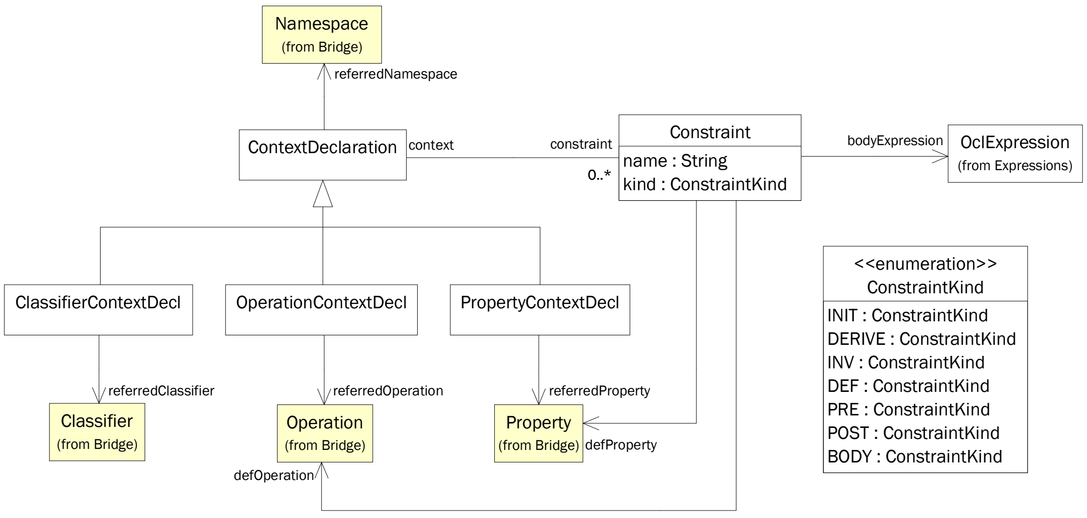


Figure 5.12: *Model of context definitions in the Kent OCL Library*

In conclusion, I do not consider the Bridge metamodel sufficient to meet the goals of this report. In particular, I wish to avoid a tight coupling between the OCL metamodel and the Pivot Model. In the Kent OCL Library, a clean separation of OCL and Bridge concepts is impossible due to the conceptual deficiencies outlined above.

5.2.3 The Definition Level

The Kent OCL Library uses an adapter-based approach for the integration of custom target languages that is very similar to the concept of model adaptation presented in Section 4.3.3. Figure 5.13 exemplifies the principle for the EMF adapter realizing the `Namespace` interface from the Bridge model. In its `lookupOwnedElement` method, this adapter simply iterates through the adapted `EPackage` and returns a corresponding adapter for the `EClassifier` with the requested name. The construction of new adapters is delegated to a central factory. My own solution essentially follows this pattern, but differs in some implementation aspects (see Section 6.2.3).

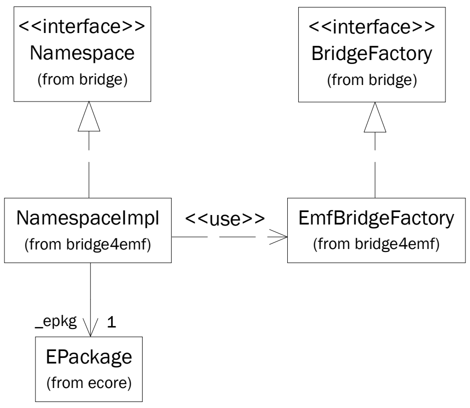


Figure 5.13: *Model Adaptation in the Kent OCL Library*

5.2.4 The Execution Level

The Kent OCL Library provides a straightforward implementation of the OCL Standard Library in Java. Each predefined OCL type is mapped to a corresponding Java type. Domain-specific representations of instance-level elements are not supported. The metaprogramming facilities are very slim. For instance, collection and tuple types cannot be accessed on the Execution Level and the `OclType` implementation is a simple wrapper around `java.lang.Class`.

Both code generation and runtime evaluation via an interpreter relies on a Visitor [GHJV95] API that is supported by each abstract syntax element. Domain-specific implementations of the `ModelImplementationAdapter` interface allow accessing properties and operations of instance-level elements in a customizable fashion. Listing 5.2 illustrates the principle with an excerpt from `EmfImplementationAdapter`. This creative approach simplifies code generation, but limits the reusability of an interpreter due to strong dependencies on the Java reflection mechanism.

```
public String getGetterName(String property_name) {
  return "get" + property_name.substring(0,1).toUpperCase() +
    property_name.substring(1,property_name.length());
}

public String getEnumLiteralValue(String enum, String enumLit) {
  return enum + ".get(\"" + enumLit + "\")";
}
```

Listing 5.2: *Adapting domain-specific execution semantics in Kent OCL*

5.3 The Epsilon Platform

5.3.1 Overview

The Epsilon project (**E**xtensible **P**latform for **S**pecification of **I**ntegrated **L**anguages for m**O**del ma**N**agement) [Epsb] was initiated at the University of York within the context of the European Integrated Project MODELWARE [MODb]. In the meantime, it has been migrated to the Eclipse GMT project (Generative Modeling Technologies) [GMT], but continues to be developed in the context of MODELWARE's successor project, MODELPLEX [MODa].

The scope of Epsilon is much larger than simply providing an OCL engine. Instead, it aims at "building a framework for supporting the construction of domain-specific languages and tools for model management tasks, i.e, model merging, model comparison, inter- and intra-model consistency checking, text generation, etc." [Epsa]. On top of a model querying language, the Epsilon platform builds several task-specific languages that share comprehensive Eclipse editing support [KPP06a]. I have included the project in my review of related work because it claims to be the first implementation of instance-level alignment of OCL with arbitrary domain-specific languages [KPP06d].

However, in contrast to the Dresden OCL Toolkit and the Kent OCL Library, Epsilon does not actually provide an OCL engine at the moment. Instead, it defines an own query language called Epsilon Object Language (EOL) [KPP06c] that is akin to, but not compatible with OCL. This has primarily practical reasons and the authors are working on model transformations to bridge the technological gap [KPP06d]. Thus, it does not pose an impediment for a conceptual evaluation within the scope of this report.

5.3.2 The Concepts and Definition Level

The Epsilon Platform does not explicitly define a common metamodel for composition with the metamodels of domain-specific languages. Instead, an abstraction layer called Epsilon Model Connectivity (EMC) specifies a number of operations which are required for an integration with the execution engine. Figure 5.14 shows a high-level overview of this architecture.

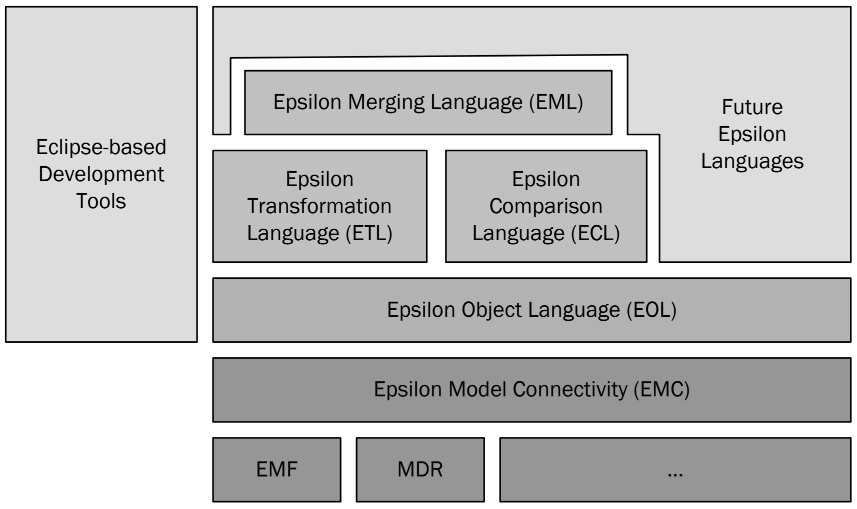

Eclipse-based Development Tools
Epsilon Merging Language (EML)
Future Epsilon Languages
Epsilon Transformation Language (ETL)
Epsilon Comparison Language (ECL)
Epsilon Object Language (EOL)
Epsilon Model Connectivity (EMC)
EMF
MDR
...

Figure 5.14: *Architecture of the Epsilon Platform (adapted from [KPP06a])*

The operations that need to be supported can be classified in two categories:

- specification of the domain-specific instantiation semantics (e.g., how to find all instances of a type)
- specification of the domain-specific semantics of the point (.) navigational operator (e.g., what does an expression `variable.property` mean)

All of these operations are declared in the interface `EolModel`. The implementations of this interface for EMF (`EmfModel`) and the Netbeans MDR (`MdrModel`) are realized in Java. Much more interesting is a special implementation called `EolM0Model` that allows for defining the above-listed semantics using EOL itself. Thus, a high-level declarative language can be used for defining the integration of an arbitrary DSL with the EOL execution engine.

Although the approach taken in the Epsilon project easily integrates different model repository technologies and permits declarative specifications of DSL semantics, some open questions remain. First and foremost, building the execution engine directly on top of the concrete syntax is disputable. Most researchers and practitioners agree that, ideally, the metamodel of a language should be the basis for the "automated, tool-supported processing of models" [SV06, p. 85]. Moreover, reducing the expressiveness to the point operator alone might not suffice to capture all possible semantics. For instance, it is unclear how the current implementation of the Epsilon platform supports operation calls, enumeration literal expressions, static features etc. At least the EMF and MDR bindings seem to be limited to property calls.

Finally, specifiying the integration with another DSL declaratively may not be adequate to actually execute a query on the System layer. In [KPP06d] the binding with a relational DSL is presented. Yet, the article does not mention how an actual database can be accessed with the EOL engine. Obviously, there is a difference between manipulating the instance of a DSL in a standard model repository like EMF and modifying one that really links to concrete System layer elements. Section 2.1.3 highlighted that this may require totally different execution semantics.

5.3.3 The Execution Level

Epsilon provides its own implementation of the OCL Standard Library, appropriately prefixing all elements with `Eol` (e.g., `EolAny`, `EolCollection`, etc.). There are no classes corresponding to `OclTuple` and `OclEnumLiteral`, though. Custom adapters for domain-specific representations of the standard types are not supported. Similar to the Kent OCL Library, all Standard Library types are simple wrappers for appropriate Java types. However, the meta level implementation is more extensive than the one in Kent OCL and additionally covers primitive types, collection types as well as user-defined types. Compared with the Dresden OCL Toolkit, this still falls short of tuple types and enum types.

Accessing model element properties is facilitated through the interfaces `PropertyGetter` and `PropertySetter`. The EMF and MDR implementations of these interfaces use their respective reflective capabilities. An additional implementation for Java employs Java reflection and alternatively tries a number of different method name patterns (see Listing 5.3).

```java
String methodName = "get" + property;
Method method = ReflectionUtil.getMethodFor(object, methodName, 0);

if (method == null){
  methodName = property;
  method = ReflectionUtil.getMethodFor(object, methodName, 0);
}

if (method == null){
  methodName = "is" + property;
  method = ReflectionUtil.getMethodFor(object, methodName, 0);
}
```

Listing 5.3: *Accessing properties of Java objects in Epsilon*

Table 5.1 summarizes the review of related work presented in this chapter. The next chapter will illustrate how the experiences and conclusions drawn from this analysis have influenced the design of my own Pivot Model.

	Dresden OCL2 Toolkit	**Kent OCL Library**	**Epsilon Platform**
Concepts Level	*CommonModel* as abstraction of MOF 1.4 and UML 1.5 containing some design flaws	*Bridge* metamodel provides central abstraction, far-reaching adjustments of the OCL type hierarchy	no dedicated common metamodel for abstracting from different DSLs
Definition Level	inheritance-based, complex mixture of object and class adapters, `ModelFacade` as central interface to foreign model repositories	model adaptation, integration of foreign repositories via adapters and *Abstract-Factory* pattern	no metamodel-based integration, model connectivity layer requiring definition of `allInstances` and dot operator semantics
Execution Level	implementation of the OCL Standard library optimized for code generator, conversion between OCL and domain-specific representation cumbersome	strictly Java-based implementation of the OCL Standard Library, domain-specific semantics through string concatenation of property accessor code	Java-based implementation of OCL Standard Library for EOL, domain semantics limited to property getter and setter abstraction

Table 5.1: *Summary of analysis of related work*

6 Results

In the previous two chapters, I have thoroughly analyzed the underlying problems arising from an integration of OCL with multiple metamodels and examined how existing implementations try to deal with these challenges. By identifying the shortcomings of previous work, I have created a solid foundation for a holistic approach to fulfilling the goals of this report. This chapter will comprehensively describe my overall solution in regards to the three-layered conceptual framework I have been using as a guideline throughout this work. It consists of two main parts. First, I will cover the design of both the Pivot Model and the metamodel integration infrastructure developed around it. This section will concentrate on the rationale behind some of the more intricate design decisions, but leave out implementation-specific details. The actual prototypical realization of the project is the topic of the second part. Here, I will particularly highlight some of the slightly more involved patterns and idioms in the implementation code. The explanations should be sufficient to provide a starting point for future extension of the components developed so far.

6.1 Realizing the Pivot Concept

6.1.1 The Concepts Level

This section presents a detailed discussion of the design of the Pivot Model. It starts with the formulation of a number of general design principles which have guided the development process. They should help to better understand some design decisions. Then, I will highlight each of the different aspects of the Pivot Model in turn and provide explanations for more controversial issues when appropriate.

Design Principles

In Section 2.3.5, I explained why Essential OCL is a good foundation for an integration of OCL with arbitrary domain-specific languages. Consequently, the Pivot Model design is based on the structure of the Core::Basic package (see Figure 2.13 for a refresher). Starting from there, I have followed the following principles:

Simplification and consolidation

My main objective was to simplify and consolidate the metamodel of Core::Basic to address the issues raised in Section 2.3.5 and Chapter 5. This includes, in particular, the problem of the missing Classifier concept. Additionally, I aimed for removing superfluous classes to flatten the inheritance tree. Also, I have removed metaclass attributes that are not relevant for evaluating OCL expressions.

Maximizing expressiveness

Core::Basic lacks some useful language capabilities which severely limits the set of valid OCL expressions. I have added those concepts that I imagine to occur frequently in domain-specific languages. Note that the model adaptation mechanism presented in Section 4.3.3 does not require a DSL to support all concepts available in the Pivot Model.

Ensuring a layered design

As pointed out earlier, I have striven to avoid any dependencies from the EssentialOCL package to the Pivot Model. This facilitates a layered design and allows splitting up the implementation of the model in separate components. Neither the Dresden OCL2 Toolkit nor the Kent OCL Library currently achieve this level of modularity.

Consistent use of multiple inheritance

Section 5.1.2 highlighted the common phenomenon of overlapping concepts in metamodels and the inconsistent use of different forms of inheritance in response to this. For instance, from a conceptual point of view there is no sound reason why TypedElement should descend from NamedElement, while MultiplicityElement is mixed in separately. The first two concepts are equally unrelated. As a result of this consideration, the Pivot Model consistently employs multiple inheritance to include these aspectual features. This yields the additional benefit of a more shallow inheritance tree. Note that this does not hinder mapping the Pivot Model to an object-oriented language without support for multiple inheritance. Firstly, code generation with mixin inheritance can significantly reduce the effort of building an implementation. Secondly, the class inheritance hierarchy may still be altered during the implementation phase to maximize code reuse in subclasses.

Keeping compatibility with Dresden OCL Toolkit

The current implementation of the Dresden OCL2 Toolkit defines additional operations on some CommonModel classes to ease the implementation of the OCL parser, type checker, and compiler. For instance, Operation adds several methods to retrieve only a subset of the owned parameters. Whenever the need for these additional operations was justified, I have included them in the Pivot Model as well.

Adjusting the OCL specification

At the moment, OCL 2.0 [OMG06c] is not yet entirely aligned with the 2.0 versions of UML and MOF. To a large extent, it is still based on the UML 1.5 standard. This is particularly evident in the OCL expressions used to define additional operations, wellformedness rules and the abstract syntax mapping. Time constraints did not allow me to update the complete specification, but Appendix A precisely defines most of the additional operations in the Pivot Model. Note that I have omitted apparently deprecated operations such as `Operation::allProperties` and `Parameter::make` [OMG06c, pp. 56].

Consistent naming

The naming of attributes and association ends in the Pivot Model follows the scheme employed by the UML 2.0 specification.

Pivot Model Design Considerations

Now that I have established some general design guidelines, I can proceed to explain the different aspects of the Pivot Model design in more detail.

Overview

Figure 6.1 gives an overview of the main concepts in the Pivot Model and their relationships. The most obvious difference to the structure of Core::Basic is that any type may possess properties and operations now. The redundant metaclasses Class and Datatype have been removed. This design alleviates most of the problems described in Section 2.3.5. In addition, it defines a clean inheritance hierarchy of the very central object-oriented concepts supported by OCL. This may serve as a guideline for the initial integration with an arbitrary domain-specific language. All of the additional metaclasses explained below only provide the means to support some of the more intricate features of the OCL concrete syntax.

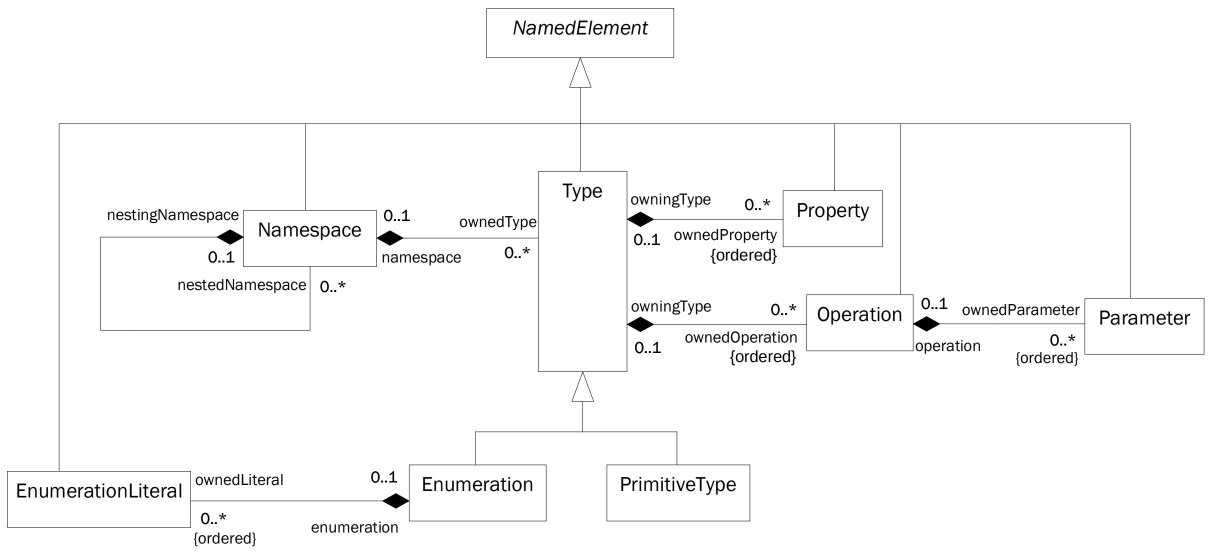


Figure 6.1: *Main concepts of the Pivot Model*

Named Elements

The Pivot Model adds a derived attribute `qualifiedName` to the abstract metaclass NamedElement (Figure 6.2). This is motivated by the current implementation of the code generator in the Dresden OCL2 Toolkit that relies on this attribute. Contrary to the definition in the UML 2.0 package Core::Abstractions:Namespaces, the qualified name is not constructed from a hierarchy of namespaces. Instead, each NamedElement has an `owner`. The exact type of the owner is subclass-specific. For instance, the owner of a Parameter is the corresponding Operation.

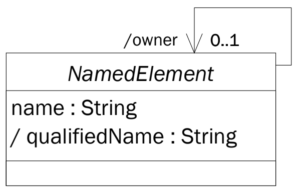


Figure 6.2: *The abstract Pivot Model class NamedElement*

Types and Typed Elements

The Type metaclass contains all of the special operations defined in the OCL 2.0 specification [OMG06c, p. 55] (Figure 6.3). To support definition constraints (see Section 2.3.4), the

Pivot Model additionally includes operations that add a new Property or Operation to a Type. These operations return the corresponding Type instance, so they can be specified as OCL queries. Elements with a type mix in the corresponding abstract metaclass TypedElement as described further above.

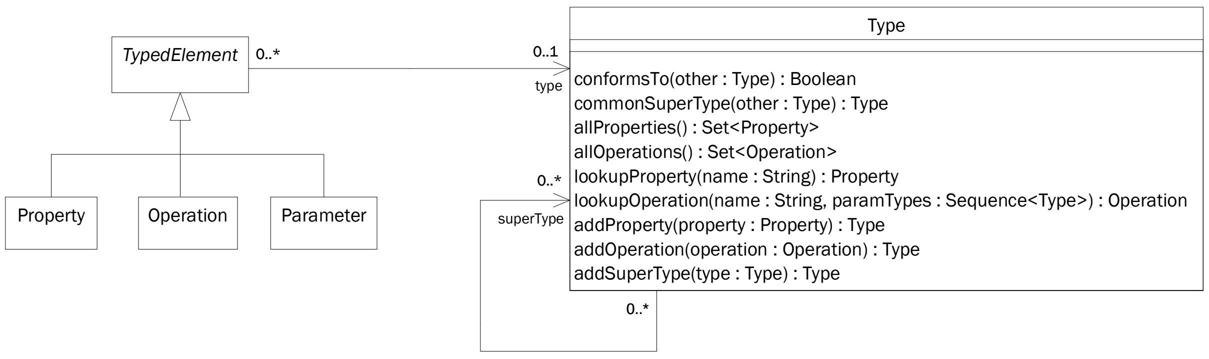

TypedElement
0..*
0..1
type
Type
conformsTo(other : Type) : Boolean
commonSuperType(other : Type) : Type
allProperties() : Set<Property>
allOperations() : Set<Operation>
lookupProperty(name : String) : Property
lookupOperation(name : String, paramTypes : Sequence<Type>) : Operation
addProperty(property : Property) : Type
addOperation(operation : Operation) : Type
addSuperType(type : Type) : Type
0..*
superType
0..*
Property
Operation
Parameter

Figure 6.3: *Types and typed elements in the Pivot Model*

Operations and Parameters

Core::Basic does not support the notion of a parameter direction kind. Since OCL defines precise semantics for passing `in`, `out`, or `inout` parameters into an Operation, I have included the corresponding meta elements from the UML Core::Constructs package (Figure 6.4).

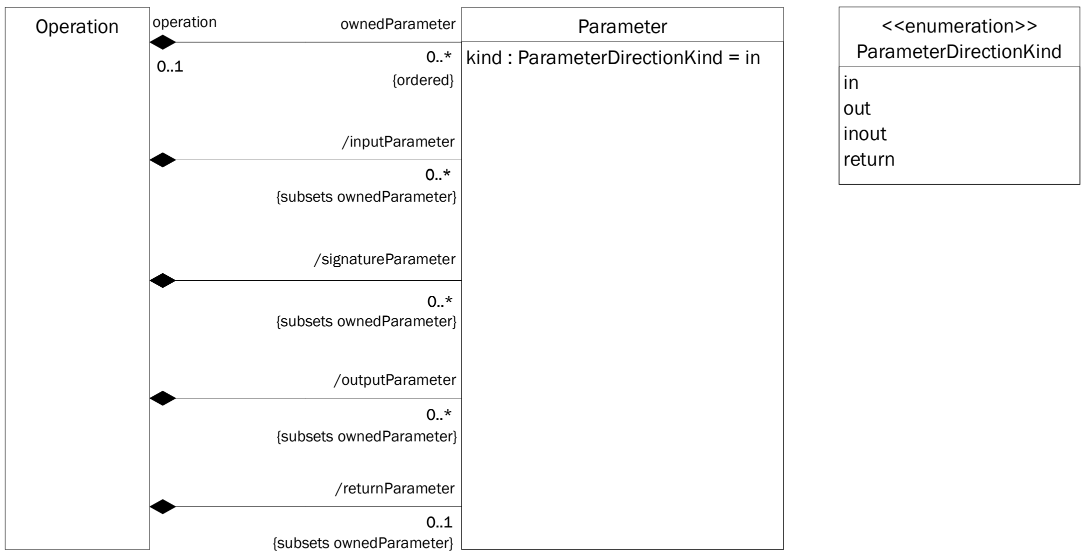


Figure 6.4: *Operations and parameters in the Pivot Model*

Multiplicity Elements

In Core::Basic, a MultiplicityElement has a lower and an upper bound. However, an OCL engine only needs to distinguish between single- and multivalued elements. The exact bounds are irrelevant. Following the design of the Dresden OCL2 Toolkit CommonModel, I have replaced the two attributes with a boolean flag `isMultiple` (Figure 6.5).

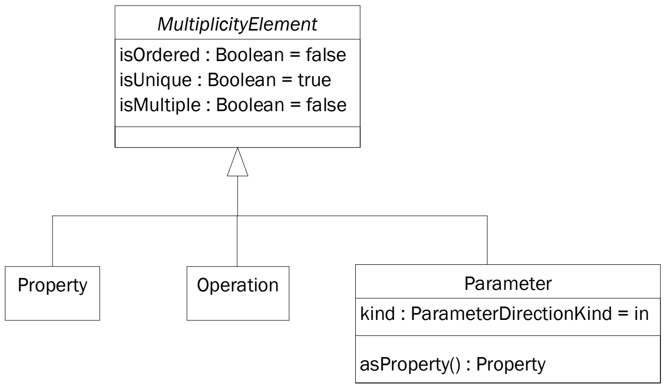

MultiplicityElement
isOrdered : Boolean = false
isUnique : Boolean = true
isMultiple : Boolean = false
Property
Operation
Parameter
kind : ParameterDirectionKind = in
asProperty() : Property

Figure 6.5: *Elements with multiplicity in the Pivot Model*

Features

Core::Basic does not know the concept of *static* properties and operations. Yet, this is a common feature in object-oriented domains and is explicitly supported by OCL. It is particularly vital for the predefined `allInstances` operation. Hence, I have added an abstraction Feature that provides a corresponding boolean flag (Figure 6.6).

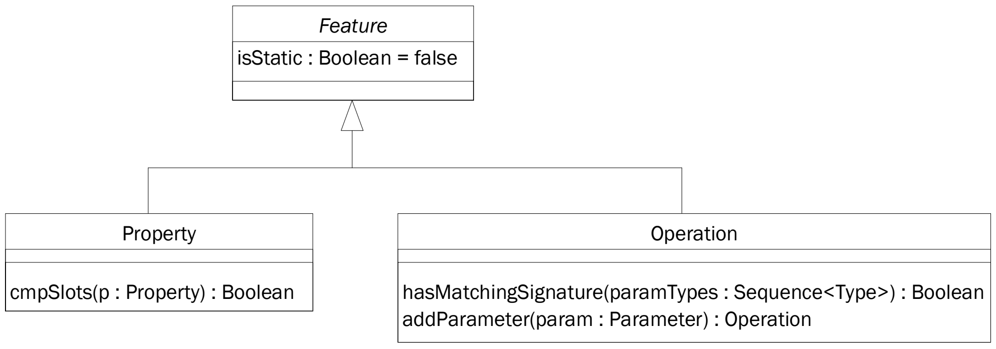

Feature
isStatic : Boolean = false
Property
cmpSlots(p : Property) : Boolean
Operation
hasMatchingSignature(paramTypes : Sequence<Type>) : Boolean
addParameter(param : Parameter) : Operation

Figure 6.6: *Supporting static features in the Pivot Model*

Primitive Types

Modeling languages usually define a set of primitive types so users do not have to specify their own. As an example, consider the Core::PrimitiveTypes package in the UML Infrastructure Library or the predefined datatypes in the EMF Ecore model. Intuitively, the arithmetic and logical operations defined in the OCL Standard Library should be available on these types as well. However, this requires a mapping from the domain-specific representation onto the corresponding OCL type.

To this end, the Dresden OCL2 Toolkit employs an additional operation `toOclType` that needs to be realized by all subtypes of Classifier. I have already mentioned this operation (and the problems associated with it) back in Section 5.1.2. The Kent OCL Library resorts to a simple `instanceof` check when creating new adapters for a domain-specific type. Both approaches introduce a dependency from the domain-specific implementations of the PrimitiveType concept to the OCL Standard library.

In the Pivot Model, I have realized a slightly different approach. The PrimitiveType metaclass defines an additional kind attribute that gives a hint to the OCL engine about the required type mapping (Figure 6.7). The default value is Unknown, which represents domain-specific data types that do not have an equivalent OCL type. For instance, EMF allows to define data types as proxies for Java classes that are not part of the model. In summary, this design not only eliminates the dependency to the OCL Standard Library, but is also flexible enough to support query languages other than OCL. If needed, the OclPrimitiveTypeKind enumeration can be extended accordingly.

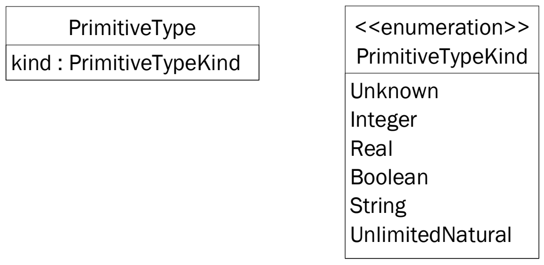


Figure 6.7: *Type mapping for primitive types in the Pivot Model*

Constraint definitions

Core::Basic does not support the notion of constraints. Thus, different forms of constraints (invariants, pre- or postconditions, etc.) cannot be expressed in the abstract syntax. Moreover, it is unclear how to add instances of OCL metaclasses to an existing model. In Essential OCL, the top-level concept ExpressionInOcl solely derives from TypedElement which cannot be added to a Namespace. In response to this problem, the Pivot Model includes a number of constraint-related classes (Figure 6.8). The model is inspired by the UML Core::Abstractions::Constraints package and the Kent OCL Bridge model presented in Section 5.2.2. Note that in contrast to the Kent OCL solution, the Pivot Model remains entirely independent of the OCL metamodel.

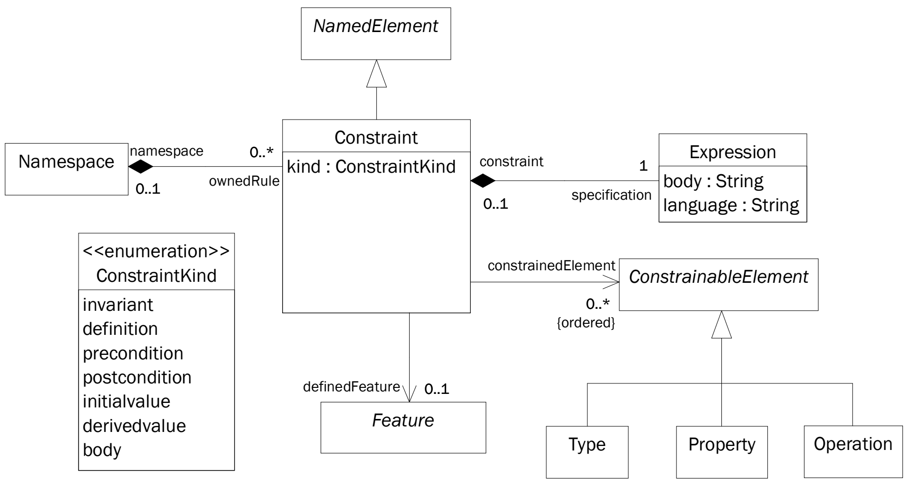


Figure 6.8: *Constraint definitions in the Pivot Model*

The model introduces an abstraction ConstrainableElement that classifies all elements that may be the target of a constraint expression. Constraints with kind definition may additionally reference the defined feature. It is worth mentioning that the UML specification denotes the context where a constraint is evaluated via an additional reference to Namespace. I have omitted this association since OCL saves an expression's context in a dedicated Variable accessible from ExpressionInOcl.

Generics

The predefined collection types in the OCL Standard Library are actually template types with the type parameter `T` [OMG06c, p. 144]. Thus, the concrete collection type `Sequence(Integer)` is created from the template `Sequence(T)` by substituting (or binding) `T` with `Integer`. Existing implementations of OCL usually create the collection types programmatically when required. The Dresden OCL2 Toolkit and the Kent OCL Library work this way. A significant drawback of this approach is that the structure of the Standard Library is buried deep inside the implementation code. Evidently, this severely impairs reusability, flexibility, and maintainability.

In this report, I propose the novel approach of modeling the Standard Library as an instance of the Pivot Model. Then, it can easily be serialized into an XMI representation and loaded into the domain-specific model when necessary (see Section 6.2.2). However, to model the operations defined on the OCL collection types, the Pivot Model needs to support templates. As an example, consider the `sum` operation in `Collection(T)` that returns the sum of all contained elements (Listing 6.1). Since the return type depends on the element type of the collection, it must be possible to specify the corresponding type parameter in the model.

```
sum() : T
post: result = self->iterate( elem; acc : T = 0 | acc + elem )
```

Listing 6.1: *The summation operation defined for Collection(T)*

However, allowing type parameters for types alone does not suffice. To see why, consider the `product` operation of `Collection(T)` which returns the cartesian product of two collections (Listing 6.2). Note that the concrete signature of this operation (in particular, its return type) not only depends on the binding of the type parameter `T`, but also on the type of the argument `c2`. This is an example for a so-called *generic operation* [Bra04]. Further note that the return type of the product operation is itself a template type, namely `Tuple(first:F, second:S)`, whose type parameters `F` and `S` are bound with the type parameters of the collection types, `T` and `T2`, respectively.

```
product(c2:Collection(T2)) : Set( Tuple(first:T, second:T2) )

post: result = self->iterate (
        e1; acc : Set(Tuple(first:T, second:T2)) = Set{} |
          c2->iterate (
            e2; acc2 : Set(Tuple(first:T, second:T2)) = acc |
              acc2->including (Tuple{first = e1, second = e2})
          )
      )
```

Listing 6.2: *The cartesian product operation of Collection(T)*

Figure 6.9 shows the model resulting from these considerations. It is based on the generics support developed for EMF 2.3 [MP07]. The model introduces a new abstraction GenericElement which classifies elements that may contain TypeParameters. A type parameter of a generic element may be *bound* with a concrete type, which means that all occurences of the parameter in the definition of the generic element are replaced with this type (Figure 6.10).

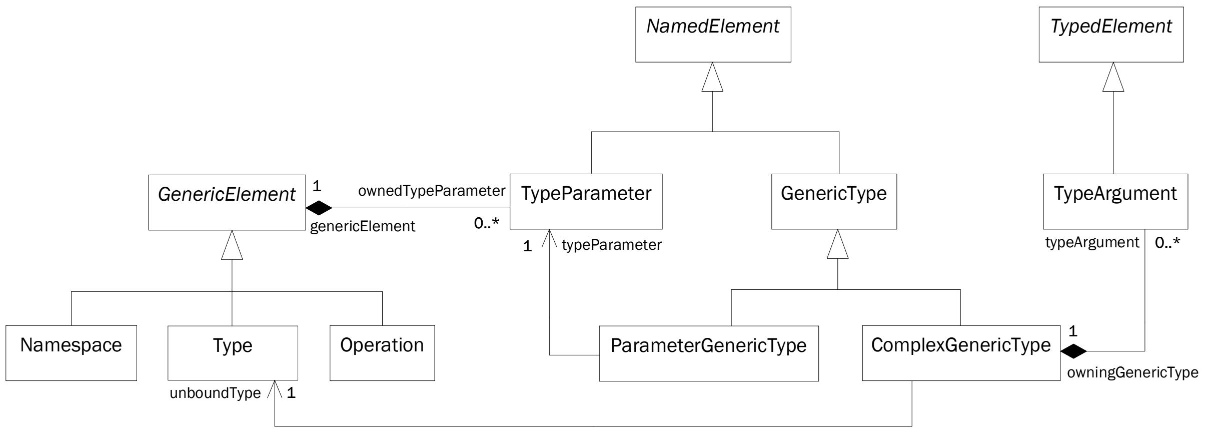


Figure 6.9: *Generics in the Pivot Model*

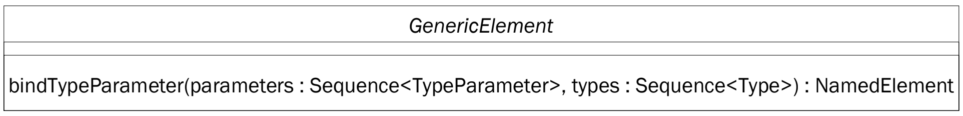

GenericElement
bindTypeParameter(parameters : Sequence<TypeParameter>, types : Sequence<Type>) : NamedElement

Figure 6.10: *Binding the type parameters of generic elements*

To actually model properties, operations or parameters with generics, a TypedElement can now alternatively reference a GenericType (Figure 6.11). Generic types exist in two flavours. A ParameterGenericType simply references a TypeParameter, as in the case of the return parameter of the `sum` operation. A ComplexGenericType, on the other hand, references a Type with unbound type parameters as well as a number of TypeArguments that will replace the type parameters during binding. In the example of the `product` operation, the return parameter has a ComplexGenericType referencing the unbound type `Tuple(first:F,second:S)` and defining two type arguments `T` and `T2`. This example shows nicely that type arguments, being plain TypedElements, can have a generic type as well. Through this design, an unlimited nesting of generic types becomes possible.

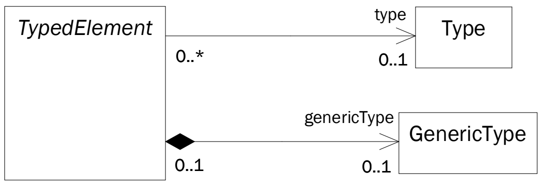


Figure 6.11: *Generic types in the Pivot Model*

It turns out that supporting generic types for typed elements does not suffice yet. Consider again the OCL collection type `Sequence(T)`. This generic type extends `Collection(T)`. Intuitively, binding type parameter `T` of `Sequence(T)` with a concrete type, say `String`, should result in `Sequence(String)` extending `Collection(String)`. Yet, the design developed so far does not cover this special case. The key observation here is that the two type parameters `T` are not the same. In fact, it is perfectly legal to label the type parameter of the sequence type with `S` instead of `T`. Correctly binding both subtype `Sequence(S)` and supertype `Collection(T)` requires `S` to be a `TypeArgument` of `Collection(T)`. This intuition leads to the introduction of a new association between Type and GenericType denoting the *generic supertypes* of a type (Figure 6.12). Then, binding a Type will cause all generic supertypes to be bound as well. If all type parameters of a generic super type are bound (i.e., it is not generic any more), it will be added to the regular superType reference list.

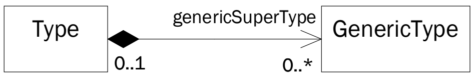


Figure 6.12: *Generic supertypes in the Pivot Model*

Integrating Essential OCL with the Pivot Model

The alignment of the Essential OCL abstract syntax model with the Pivot Model does not pose any problems. Basically, it involves replacing all references to EMOF with the corresponding Pivot Model elements. The resulting package diagram is shown in Figure 6.13. Note that for a complete alignment, the static semantics (i.e., the wellformedness rules), the abstract syntax mapping as well as the dynamic semantics need to be adapted as well. Within the bounds of this report, I have not yet achieved this level of completeness, but the necessary adjustments should be equally straightforward.

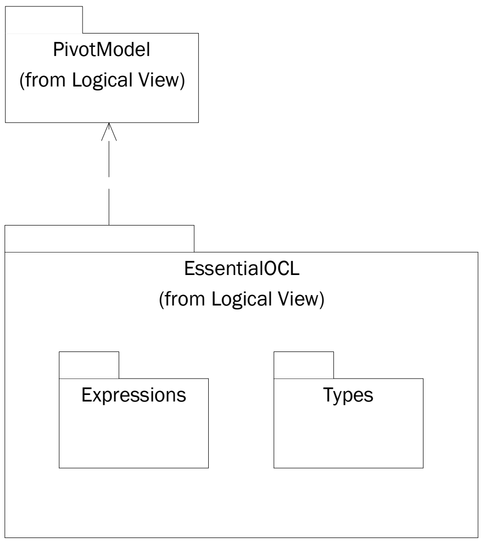


Figure 6.13: *The adjusted Essential OCL metamodel depends on the Pivot Model*

To a large extent, the Essential OCL metamodel created for this report corresponds to the one given in the specification [OMG06c, pp. 172]. However, I have amended the abstract syntax in a few places when either the alignment with the Pivot Model necessitated this or it appeared to be useful for increasing the expressiveness of the language. In the following, I will present only these special cases. Appendix B contains the full specification of the adapted metamodel.

To start with, the simplified type hierarchy in the Pivot Model results in a flattened inheritance tree in the OCL Types package (Figure 6.14). This is a clear improvement over the elaborate design of the Kent OCL Library that was shown in Figure 5.11.

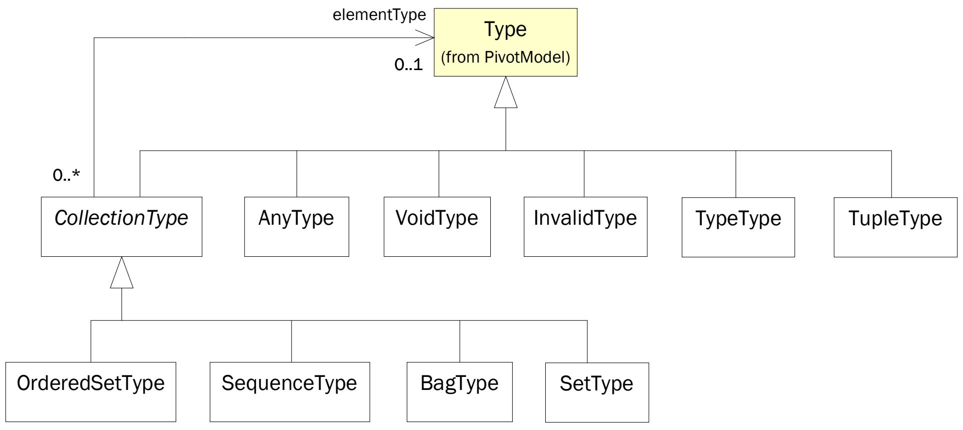


Figure 6.14: *The Essential OCL Types package aligned with the Pivot Model*

Furthermore, I have removed the meta class NavigationCallExp as super class of PropertyCallExp (Figure 6.15). In Complete OCL, a NavigationCallExp refers to association ends and association classes. Obviously, these UML-specific references are not required for a mapping to arbitrary domain-specific languages. However, the Essential OCL specification only suppresses the property NavigationCallExp::navigationSource stating that it is "not needed in this context" [OMG06c, p. 171]. Paradoxically, the now useless metaclass itself is left in the model. Removing it results in a more concise design.

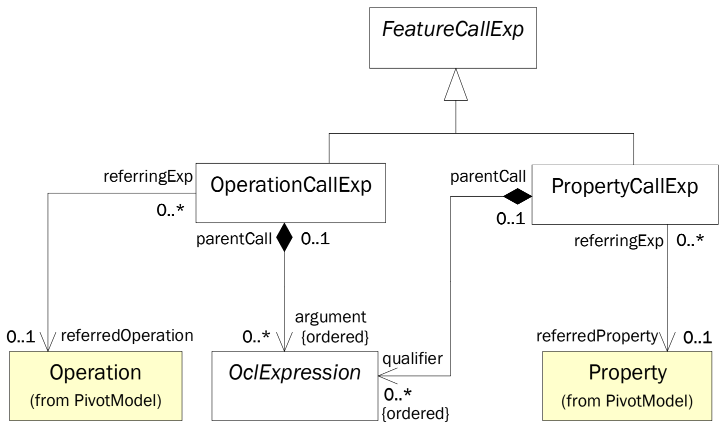


Figure 6.15: *Feature call expressions in the aligned Essential OCL*

Also shown in Figure 6.15 is another deviation from the specification. I have added the association PropertyCallExp::qualifier from Complete OCL because, in my opinion, qualified property access is a very useful language concept to support, for instance, properties representing hashtables. Since an OCL parser simply stores the qualifiers in the abstract syntax instance of an OCL expression, a DSL does not even need to contain the UML concept of qualified association ends. Providing suitable runtime semantics already suffices (see Section 6.1.3 for more details).

Next, I have renamed the meta class TypeExp to TypeLiteralExp and introduced an inheritance relationship to LiteralExp (Figure 6.16). To explain the motivation for this change, I will briefly discuss the concept of type expressions, which indeed is inadequately defined in the OCL 2.0 specification. To start with, a TypeExp is specified as "an expression used to refer to an existing meta type within an expression. It is used in particular to pass the reference of the meta type when invoking the operations oclIsKindOf, oclIsTypeOf and oclAsType" [OMG06c, p. 43]. Unfortunately, there is no production rule in the OCL grammar and no abstract syntax mapping for TypeExp. Thus, the representation of type references remains unspecified. This lack of precision has prompted alternative solutions such as the introduction of a special OclOperationWithTypeArgExp in the Dresden OCL2 Toolkit which represents the operations `oclIsKindOf`, `oclIsTypeOf` and `oclAsType`. This approach has a significant disadvantage, though. It introduces specifics of the Standard Library (i.e., the names of these operations) into the definition of the abstract syntax and, consequently, into the grammar used to build the parser.

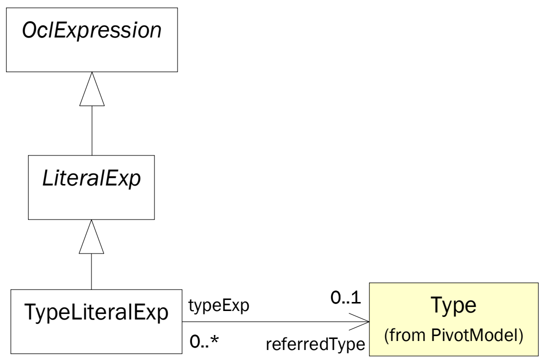


Figure 6.16: *Model of the TypeLiteralExp in Essential OCL*

My understanding of the type concept in OCL is as follows: Referring to types in OCL expressions is syntactically the same as referring to enumeration literals, namely, using a path name literal. Since EnumLiteralExp subclasses LiteralExp, a consistent class hierarchy should define the same relationship for type expressions. This intuition results in the changes to the metamodel outlined in figure 6.16. The production rules in the OCL grammar need to be amended as follows:

```
OclExpressionCS   ::= LiteralExpCS
LiteralExpCS      ::= TypeLiteralExpCS
TypeLiteralExpCS ::= pathNameCS
pathNameCS        ::= simpleNameCS ('::' pathNameCS)?
```

Now, an interesting observation directly following this definition is that a TypeLiteralExp can appear anywhere in an OCL term. Consequently, instead of

```
self.property.oclIsTypeOf(MyNamespace::MyType)
```

one could also write

```
let myTypeVariable = MyNamespace::MyType in
   self.property.oclIsTypeOf(myTypeVariable)
```

This expression highlights the reflective capabilities of OCL and is comparable to defining a variable of type `java.lang.Class` in Java. It appears that the OCL specification has originally envisaged this usage because it is directly supported by the type system. The variable `myTypeVariable` in the expression above has the type `OclType`, which is the singleton instance of the meta class TypeType [OMG06c, p. 140]. Despite the conceptual clarity, this arrangement also entirely alleviates the need for the OclOperationWithTypeArgExp added in the current Dresden OCL2 Toolkit. Instead, a standard OperationCallExp may represent the predefined type checking operations since it takes any OclExpression as an argument, in particular also a TypeLiteralExp.

Finally, I have renamed NullLiteralExp to UndefinedLiteralExp (Figure 6.17). Regarding this metaclass, the OCL specification is particularly ambiguous. In addition to missing grammar production rules and abstract syntax mapping, there is not even an explanation in the *Abstract Syntax* chapter. The most obvious meaning, however, is that this expression refers to the OCL equivalent of a `null` value. In OCL, `null` is defined as the single instance of the Standard Library type `OclVoid` which itself is the singleton instance of the meta type VoidType [OMG06c, p. 138]. Unfortunately, there is no agreement on how this literal looks like. While Section 11.2.3 of the specification simply defines it as '`null`', practically all wellformedness rules use the string '`OclUndefined`' instead. In the authoritative OCL 2.0 book, Warmer and Kleppe employ '`undefined`' when discussing the topic [WK03, p. 163]. I have decided to follow this suggestion since it agrees best with the naming of the single `OclInvalid` instance called '`invalid`'. As a result, I have renamed the corresponding literal expression to UndefinedLiteralExp. The OCL grammar should be extended with

```
LiteralExpCS          ::= UndefinedLiteralExpCS
UndefinedLiteralExpCS ::= 'undefined'
```

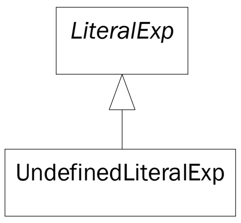


Figure 6.17: *The metaclass UndefinedLiteralExp in Essential OCL*

6.1.2 The Definition Level

As outlined in Section 4.3.3, I have chosen *model adaptation* for composing the metamodel of OCL with arbitrary domain-specific languages. Figure 6.18 exemplifies the principle for the Pivot Model Type concept. The actual implementation differs in some minor respects (see Section 6.2.2), but the core idea of using an *Object Adapter* [GHJV95] is essentially the same.

The workings of the adapter mechanism should be mostly self-explanatory. A code generation facility such as EMF automatically creates suitable getter methods for the attributes and associations of the Pivot Model element Type. All operation calls on the corresponding `Type` interface are simply forwarded to the adapted concept in the foreign DSL. For instance, a call to `getName()` will be forwarded to the adapted `EClass`'s `getName` method. Similarly, calling `getOwnedProperty()` will result in a list of `Property` adapters for the `EStructuralReferences` contained in the adapted `EClass`.

Obviously, the problem of adapting an arbitrary DSL for usage with OCL reduces to defining a suitable mapping (as observed in Section 4.1) and implementing the adapters for the corresponding Pivot Model interfaces. Figure 6.19 shows a possible mapping for the Ecore metamodel using a simplified notation that relates the Pivot Model interfaces to the corresponding concepts in the target DSL. Remember that this represents an integration in *Model Space* (see Section 4.2). Of course, a similar approach is possible for metamodels defined on M2, with execution semantics

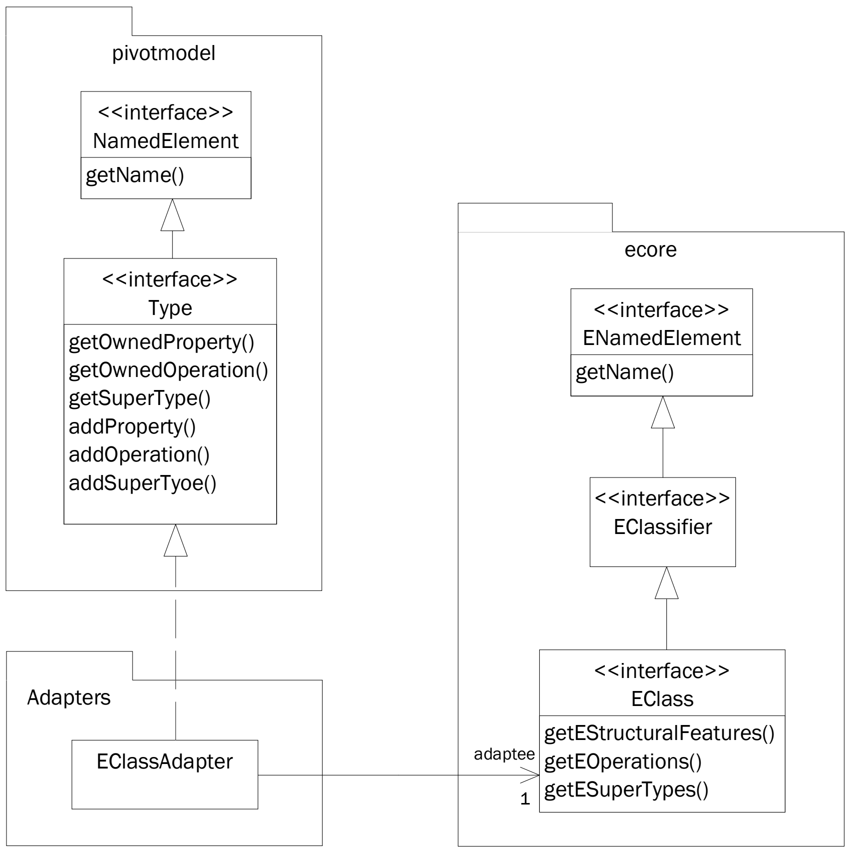


Figure 6.18: *Adapting an Ecore EClass for the Pivot Model Type concept*

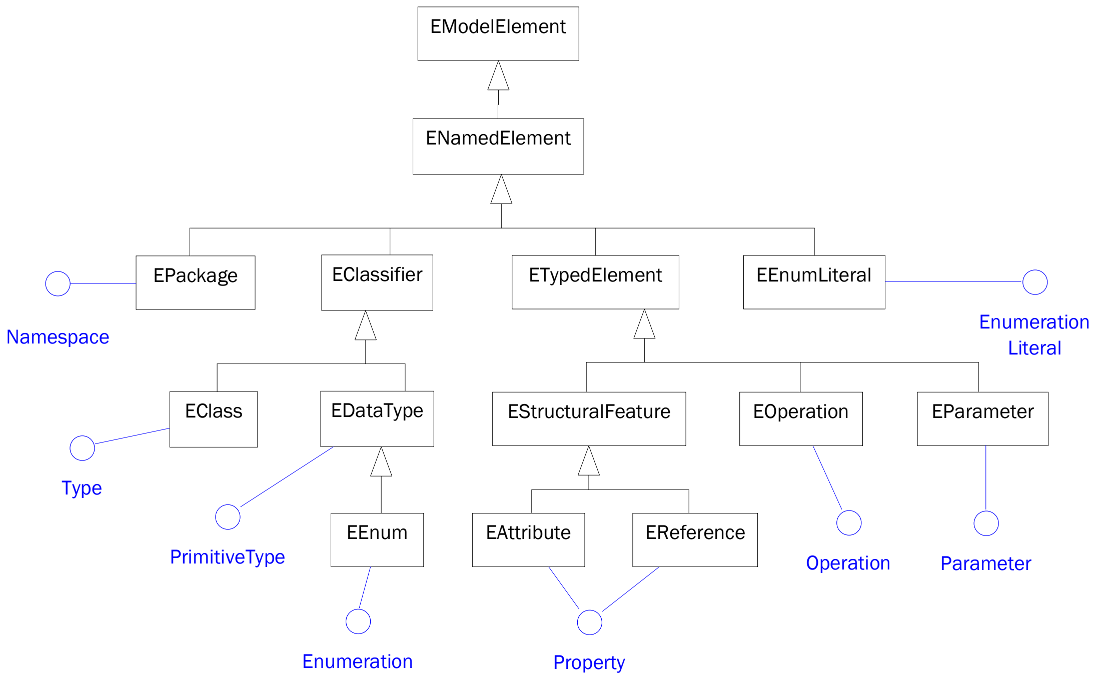


Figure 6.19: *The Ecore metamodel adapted to the Pivot Model*

in *System Space*. Figure 6.20 depicts the mapping of the example language PML required for evaluating the constraints presented in Section 4.1.

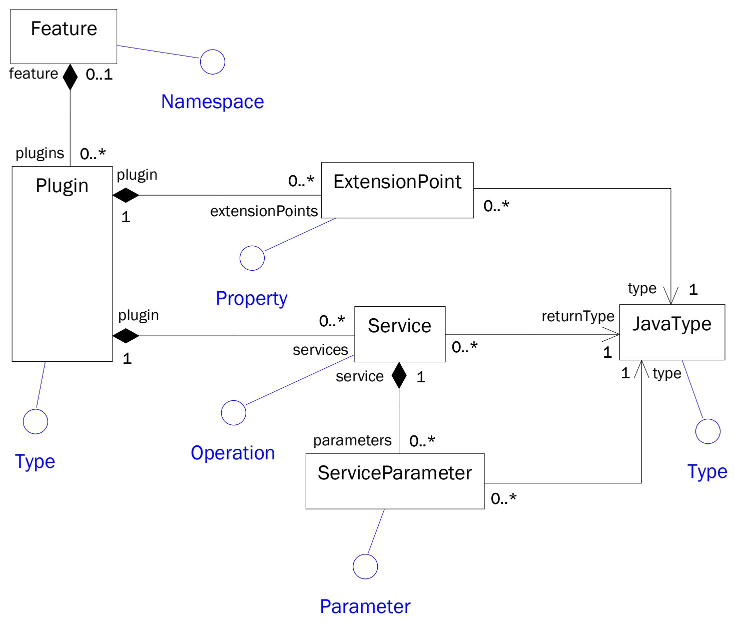


Figure 6.20: *PML adapted to the Pivot Model*

So far, I have only described the adaptation of the *structure* of a foreign DSL. In Section 4.3.3, I claimed that the model adaptation approach is equally useful for adapting foreign model repository technologies. This is immediately obvious from the fact that the Pivot Model adapters may additionally include logic to access custom model repositories. However, in a sound design the structural adaptation should be kept separate from the repository adaptation. To this end, different solutions are imaginable. For instance, a common base class for domain-specific adapters that rely on the same repository technology could provide the necessary abstraction. Another possibility is to use the *Chain of Responsibility* pattern [GHJV95] and create an adapter chain where structural adaptation occurs after repository adaptation. Finally, following the idea from the Epsilon platform (see Section 5.3.2), a special form of structural adapter may allow specifying the adaptation logic in a higher-level visual or declarative language. A first step towards this goal is to annotate the elements of a target DSL to define the mapping to the concepts of the Pivot Model. A code generator can then use this information to automatically generate most of the adapter implementation. Section 6.2.2 will provide a brief outline how to implement this approach in EMF.

Lastly, I would like to draw attention to one more useful property of the model adaptation mechanism. The key observation is that, in addition to merely adapting the foreign repository, the adapters provide a way to add transient elements to the model. As an example, consider OCL expressions that are defined in external files. An OCL parser needs to include the corresponding `Constraint` instances in the `ownedRule` reference list of the `Namespace` that represents the package declared for these expressions. Another typical application area are constraints that *define* new properties and operations on types. Obviously, these features need to be present in the model when parsing other expressions referencing them. Yet, in most cases it will be undesirable to alter the contents of the foreign repository directly. In fact, the semantics for adding new elements may be entirely undefined or, even worse, unsupported. Strictly-speaking, an OCL parser should not add new elements to the model at all, because according to Section 2.1.1, the textual OCL expressions already represent one part of the model. The adapter-based approach provides a solution to this seemingly contradictory problem. The OCL abstract syntax instances created by a parser are simply stored inside the adapters. Then, querying the `ownedRule` of a `Namespace` or the `ownedOperation` of a `Type` yields the union of these transient elements and the adapters created for the objects in the foreign model repository.

6.1.3 The Execution Level

In Section 5.1.4, I highlighted the problems in the Standard Library implementation of the Dresden OCL2 Toolkit that hinder its integration with arbitrary domain-specific languages and dynamic execution via an OCL interpreter. Summarized, I had identified the following issues:

1. complex and inconsistent adaptation of domain-specific types to OCL types
2. lacking capability to reflectively call operations on predefined types
3. unnecessary explicit target type specification when retrieving properties
4. missing support for passing `OclType` objects as parameters to operations

To address the first problem, I propose defining the entire Standard Library as a set of interfaces rather than the elaborate and hard-to-understand class hierarchy that exists at the moment. Apart from the fact that this is a fundamental object-oriented design principle, the use of interfaces allows for arbitrary (hence, domain-specific) implementations of the Standard Library concepts. Thus, for one DSL an OCL Integer may map to a `java.lang.Long`, for another one it might represent an utterly different type.

To convert between the different representations, I suggest an adapter-based approach similar to the one used on the Definition Level. Quite simply, a top interface `OclAdapter` with a single method `getAdaptee` may provide the necessary abstraction. Through this design, converting OCL types back to their domain-specific representation (as required when passing parameters to operations) is localized in the corresponding adapters. Compared with the current approach of using a central factory class which does all the conversions, my solution provides a significantly better *separation of concerns*.

Of course, creating OCL representations for domain-specific types that are returned by property or operation calls is not as simple because it requires knowledge about the domain. However, a practical (and extensible) approach is to manage the corresponding mappings in a central adapter lookup table. Ideally, this table can be configured via a declarative mechanism. For an Eclipse integration, an obvious solution would be to define a suitable extension point. Within Eclipse, an even simpler approach is to use the built-in adapter framework. It allows "translating one type of object into a corresponding object of another type" [CR06, p. 714] and is loosely based on the *Extension Objects Pattern* [Gam96]. The core idea is to query whether an object realizes the interface `IAdaptable`. In this case, an adapter for a certain type (i.e., the OCL type) can be retrieved via the `getAdapter` method. Otherwise, an `IAdapterManager` (configured through an extension point) can be asked to provide the adapter instead.

The second of the identified problems can be easily solved by moving the reflective operations into the `OclRoot` interface. To make the semantics of the operations more intuitive, I suggest renaming them from `getFeature` to `getPropertyValue` and `invokeOperation`. To address the third issue, I propose adding operations to retrieve properties as one of the predefined collection types. As described in Section 5.1.4, this eliminates the need for a dedicated type parameter. Also, an operation is required that allows passing one or several qualifiers to provide the runtime semantics for the corresponding abstract syntax elements.

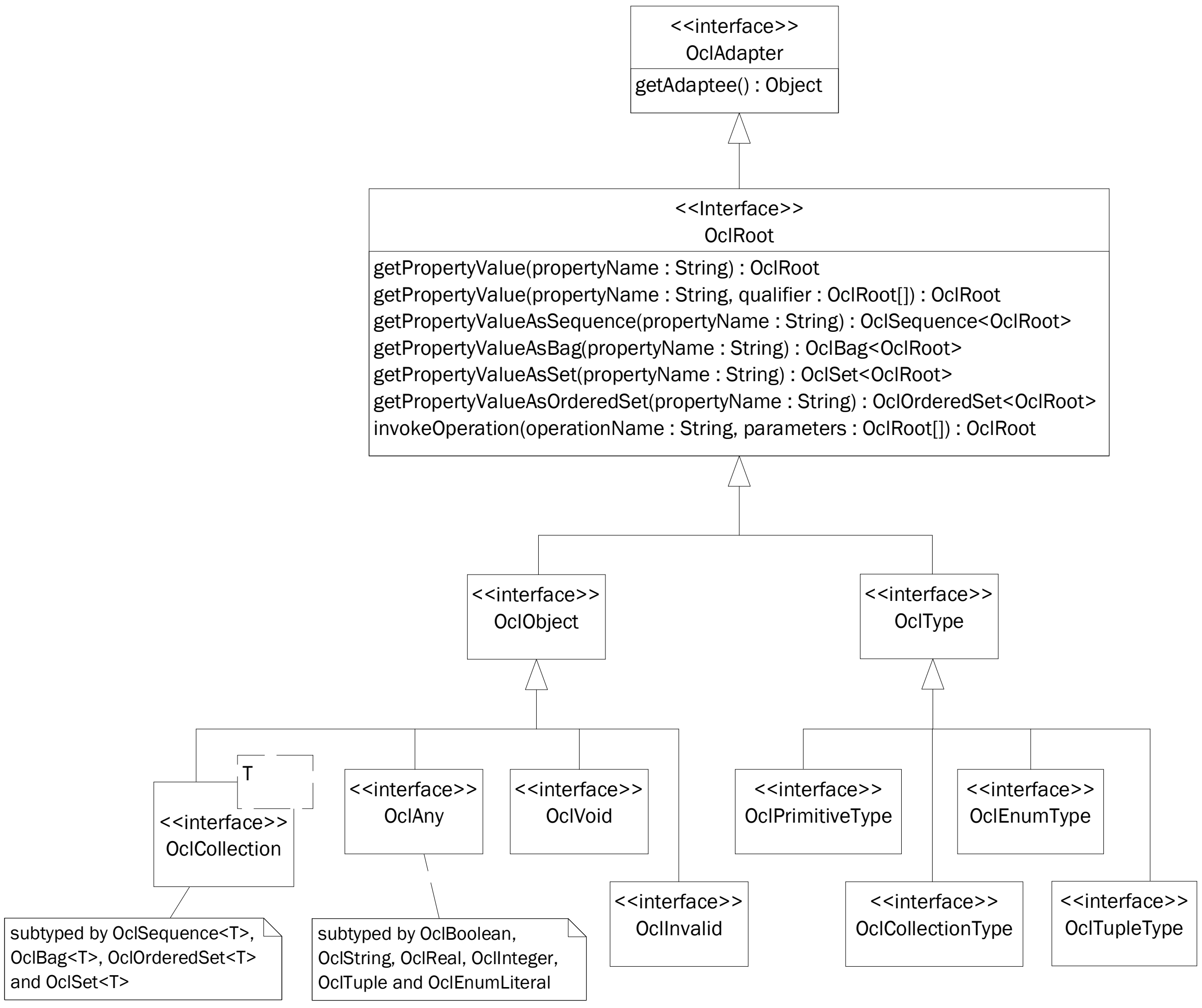

<<interface>>
OclAdapter
getAdaptee() : Object
<<Interface>>
OclRoot
getPropertyValue(propertyName : String) : OclRoot
getPropertyValue(propertyName : String, qualifier : OclRoot[]) : OclRoot
getPropertyValueAsSequence(propertyName : String) : OclSequence<OclRoot>
getPropertyValueAsBag(propertyName : String) : OclBag<OclRoot>
getPropertyValueAsSet(propertyName : String) : OclSet<OclRoot>
getPropertyValueAsOrderedSet(propertyName : String) : OclOrderedSet<OclRoot>
invokeOperation(operationName : String, parameters : OclRoot[]) : OclRoot
<<interface>>
OclObject
<<interface>>
OclType
T
<<interface>>
OclCollection
<<interface>>
OclAny
<<interface>>
OclVoid
<<interface>>
OclInvalid
<<interface>>
OclPrimitiveType
<<interface>>
OclEnumType
<<interface>>
OclCollectionType
<<interface>>
OclTupleType
subtyped by OclSequence<T>, OclBag<T>, OclOrderedSet<T> and OclSet<T>
subtyped by OclBoolean, OclString, OclReal, OclInteger, OclTuple and OclEnumLiteral

Figure 6.21: *An improved Standard Library design for the Execution Level*

Lastly, the solution to the problem of `OclTypes` as parameters of type-checking operations (e.g., `oclIsKindOf` or `oclAsType`) is to unify the inheritance hierarchy of the Standard Library by making `OclType` descend from `OclRoot`. Through this design, `OclType` also reuses the reflective capabilities described above. This is required to call static operations such as the predefined `allInstances` operation. Figure 6.21 summarizes all these considerations in a model of an improved Standard Library design.

So far, I have only described amendments to the current Dresden OCL2 Toolkit Execution Level design. In addition, I will briefly outline an extension required for an envisaged QVT engine implementation. In a model transformation scenario, querying property values alone does not suffice. It must also be possible to set properties or add values to a multi-valued property. To this end, the QVT specification defines two different concrete syntax forms for an assignment expression (see Listing 6.3).

```
mysimpleproperty := "hello";
mymultivaluedproperty += object Node {...}; // additive semantics
mymultivaluedproperty := object Node {...}; // reset list and re-assign
```

Listing 6.3: *Property assignment in QVT*

To support assignments on the *Execution Level*, I suggest introducing an additional interface `MutableOclRoot` that may be mixed in by adapters of domain-specific elements whose properties can be changed. Figure 6.22 shows the operations required for supporting the different assignment semantics.

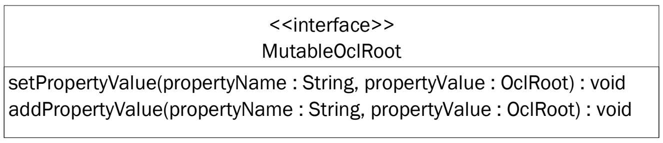

<<interface>>
MutableOclRoot
setPropertyValue(propertyName : String, propertyValue : OclRoot) : void
addPropertyValue(propertyName : String, propertyValue : OclRoot) : void

Figure 6.22: *Supporting property assignments for model transformations in QVT*

6.2 Prototypical Implementation

6.2.1 The Concepts Level

Initially, I created the Pivot Model in *Rational Rose Modeler* [IBM]. An import into the Eclipse Modeling Framework yielded a corresponding Ecore model. The EMF code generator then produced Java interfaces and implementation classes from this model. Additionally, I used the framework to create basic user interface support for displaying instances of the model in a simple tree viewer. This included a number of `ItemProvider` classes (responsible for providing labels, icons and property descriptors for each model element) as well as a set of default icons. Of course, the generated implementation was only the starting point for extensive adaptation and extension. For instance, the additional operations defined by the OCL specification and the logic for determining the value of derived properties had to be implemented by hand. Also, the visual appearance of Pivot Model elements in the UI required major customizations. For this purpose, EMF supports merging generated with user-provided code, so manual changes are not lost upon regeneration.

In addition to the Pivot Model, I also extended the Essential OCL model provided by the *SmartQVT* project [BD07] to integrate it with the Pivot Model and include the enhancements described in Section 6.1.1. In that section, I showed how a careful design helps avoiding circular dependencies between the Pivot Model and the Essential OCL metamodel. During implementation, this facilitated separating the code generated for the two metamodels in different Eclipse plug-ins thereby improving encapsulation, maintainability and reusability. In the following, I will highlight a few aspects of the EMF implementation of the Pivot Model and Essential OCL metamodel that emerged during development.

Resolving multiple inheritance

EMF provides two different ways for specifying how to resolve multiple inheritance in the generated Java code. One possibility is to use a custom stereotype «extend» to mark the realization relationship that should be generated using implementation inheritance. Figure 6.23 exemplifies this approach for the Essential OCL Variable element. For this model, EMF will generate interfaces for all three metaclasses as well as an abstract class `NamedElementImpl` (providing the implementation for `getName`) which is extended by a concrete class `VariableImpl`. The `getType` operation declared by the `TypedElement` interface is mixed into `VariableImpl` as well. The disadvantage of this mechanism is that nevertheless an abstract class `TypedElementImpl` is created although `VariableImpl` cannot extend it.

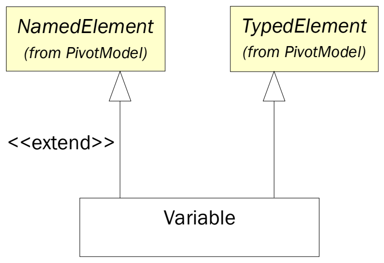


Figure 6.23: *Resolving multiple inheritance in EMF with the «extend» stereotype*

As a result, I have employed the second method of resolving multiple inheritance in EMF. For this approach, some abstract metaclasses have to be explicitly denoted as interfaces using the «interface» stereotype. Then, EMF does not generate an (abstract) implementation class. This allows for a much more fine-grained control of the code generation process. As hinted in Section 6.1.1, I have also slightly altered the inheritance hierarchy to maximally exploit implementation inheritance. Figure 6.24 illustrates the result for the Pivot Model Operation element. The adapted design particularly benefits the customization of the `ItemProvider` classes mentioned above since they are generated using the same inheritance hierarchy as the model classes. Thus, the logic for creating labels and updating the UI can be effectively shared.

Hiding EMF dependencies

By default, EMF generates code that has dependencies to the EMF API. For instance, multi-valued attributes have the type `EList` and interfaces created from the model elements extend `EObject`. Configuring the code generator with the following settings produces a "clean" API without any visible dependencies to EMF:

- Suppress EMF Metadata: true
- Suppress EMF Model Tags: true
- Root Extends Interface: empty
- Suppress EMF Types: true

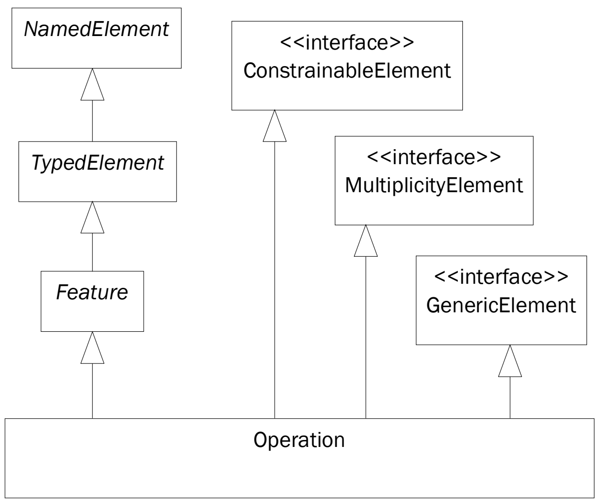

NamedElement
<<interface>> ConstrainableElement
TypedElement
<<interface>> MultiplicityElement
Feature
<<interface>> GenericElement
Operation

Figure 6.24: *Resolving multiple inheritance in EMF by explicitly denoting interfaces*

This represents a major advantage over a JMI-based implementation since clients of the Pivot Model API now remain totally oblivious of the underlying technology.

Providing missing datatypes

The definition of the Pivot Model employs a number of primitive datatypes (e.g., String and Boolean) as well as some of the OCL collection types (Sequence, Set). As outlined in Section 2.3.4, these datatypes are instances of metaclasses on a higher meta layer. Since the Pivot Model in conjunction with the Essential OCL metamodel forms a reflexive metamodel, the required data types must be provided by a bootstrapping process. In EMF, this can be achieved through the definition of EDataTypes that are mapped to suitable Java classes. Figure 6.25 shows how this is modeled in Ecore. Note that for consistency reasons, I have provided a full set of datatypes including those that are not actually used in the specification of the Pivot Model. Further note that the definition of generic datatypes using the «parameter» stereotype is an EMF 2.3 feature not supported in previous releases.

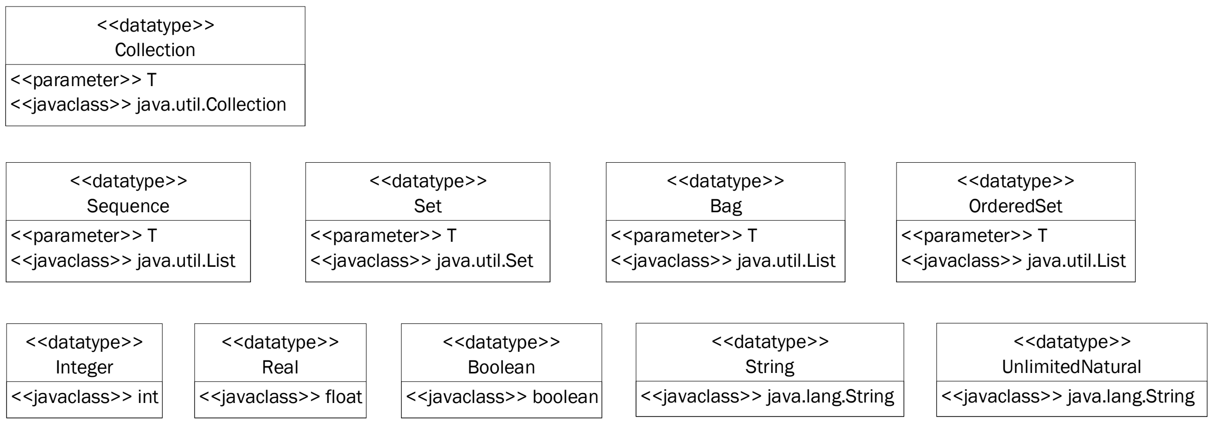


Figure 6.25: *Ecore model of datatypes used in the definition of the Pivot Model*

Type evaluation of OCL expressions

In OCL, each expression has a well-defined type that is determined based on wellformedness rules given in the specification. The current Dresden OCL2 Toolkit implementation uses the *Visitor* design pattern [GHJV95] to walk the abstract syntax tree and evaluate the type of each sub-expression. In this project, I have chosen a different approach and implemented the wellformedness rules directly inside the various `OclExpression` subclasses by overriding the `getType` method inherited from `TypedElementImpl`. Thus, the type evaluation is deferred until it is actually required (e.g., during execution via an OCL interpreter or code generator). This does not only speed up the parsing process, but also allows to parse "incomplete" expressions, which can be a useful feature during iterative development. Furthermore, the necessary logic is localized in the corresponding expression rather than being cumulated in a single Visitor class thereby fostering separation of concerns. Obviously, my approach fails to check the validity of an entire OCL expression immediately after parsing. Yet, this disadvantage can easily be alleviated by implementing a Visitor that simply tries to retrieve the type of all expression nodes. Then, it is even possible to "collect" several error messages on the way and present a summarizing report to the user rather than canceling the process when encountering the first problem.

Providing an OCL parser

For testing purposes and to provide a solid foundation for future work, it was necessary to be able to parse OCL expressions. However, the creation of a real OCL parser or the adaptation of an existing one are far outside the scope of this project. For this reason, I have devised an alternative concrete syntax for OCL which builds on XML and, thus, eliminates the need for a dedicated parser. I have coined the new language *XOCL* (XML-based OCL). Listing 6.4 shows one of the two wellformedness rules defined for the example language PML in the introductory example in Section 4.1. As a quick reminder, the `body` attribute (line 3) contains the complete expression as a string comment in OCL syntax.

```
 1 <xocl:NamespaceXS pathName="pml">
 2   <ownedRule name="idNotEmpty" kind="invariant" constrainedElement="Plugin">
 3     <specification body="self.id->notEmpty()">
 4       <bodyExpression xsi:type="xocl:CollectionOperationCallExpXS"
 5                       referredCollectionOperation="notEmpty">
 6         <source xsi:type="xocl:PropertyCallExpXS" referredPropertyName="id">
 7           <source xsi:type="xocl:VariableExpXS"
 8                   referredVariable="//@ownedRule.0/@specification/@context"/>
 9         </source>
10       </bodyExpression>
11     </specification>
12   </ownedRule>
13 </xocl:NamespaceXS>
```

Listing 6.4: *Excerpt from PML wellformedness rules in XOCL*

Note that each expression name in XOCL is postfixed with `XS` which stands for *XML Syntax*. This naming scheme is inspired by the EBNF definition of the OCL concrete syntax which adds a postfix `CS` to "clearly distinguish between the concrete syntax elements and their abstract syntax counterparts" [OMG06c, p. 61].

Most of the XOCL elements have direct correspondents in the abstract syntax. In some cases, however, I had to introduce additional elements to account for the insufficiencies of the XML syntax. Consider the expression type `CollectionOperationCallExpXS` (line 4 in Listing 6.4) as an example. In the OCL concrete syntax, collection operations are differentiated from "regular" operation calls by using the '->' instead of the '.' (dot) operator. This distinction is not possible in XOCL. Yet, it is strictly required for this particular expression because it represents the special case of using a single object (the `id` property) as an implicit collection. Introducing a dedicated expression type provides the necessary means for the parser to identify this situation.

Lastly, notice the reference to the `context` variable (representing the contextual type of an OCL expression accessible via the `self` identifier) in line 8 of Listing 6.4. Apparently, the referenced variable is not part of the XML file. This is because it is deduced from the context declaration (the attribute `constrainedElement` in line 2). I have implemented a simple parsing algorithm that automatically creates the necessary variables of an ExpressionInOcl based on the format of the context declaration. As an example, the string "`MyType::op(param:MyType):MyType`" which denotes an operation (e.g. for the definition of pre- or postconditions) will result in the creation of the variables `self`, `param` and `result`, all of type `MyType`.

Now, an interesting observation is that the data model of XOCL strongly resembles the Essential OCL metamodel. In fact, the only major difference is that all references (e.g., types, properties or operations) are simple strings identifying the referred element via its name or path name. This gives rise to the idea of modeling XOCL in Ecore by adapting the existing Essential OCL model and including the few missing elements from the Pivot Model (Namespace and Constraint). The main benefit of this approach is that the XML serialization and deserialization of an XOCL expression can be fully delegated to EMF. The task of building an XOCL parser thus reduces to the implementation of a Visitor that walks the XOCL abstract syntax tree and creates the corresponding instances of the Essential OCL abstract syntax on the way. Appendix C contains the complete specification of XOCL as a set of UML diagrams.

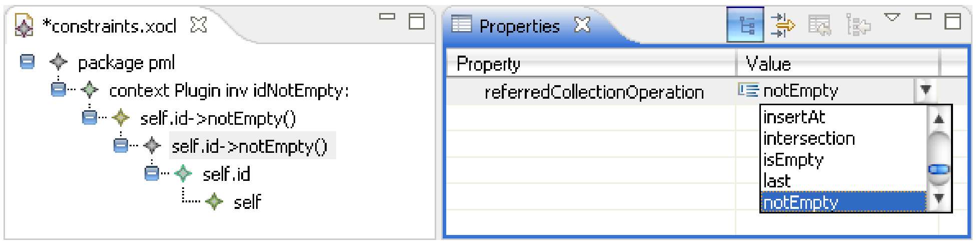


Figure 6.26: *Visually creating XOCL expressions in an editor*

Finally, using EMF yields the added benefit of an automatically generated editor, so an XOCL expression can be assembled visually. Figure 6.26 illustrates how the XOCL editor displays the example expression from Listing 6.4. Note that I have considerably adapted the visual representation of XOCL elements in this editor, so that the rendering resembles the OCL concrete syntax. The parser itself is realized as an implementation of the `IOclParser` interface (Figure 6.27). This abstraction allows reusing other UI components developed for this project (e.g., a wizard for importing OCL expressions) for a future implementation of a "real" OCL parser.

Designing an integration framework

All components developed in the course of this project are integrated into a simple MDSD and OCL infrastructure which I have coined *Model Bus*. The design has been inspired by ideas presented in [Wen06b] but additionally reflects the conceptual framework used as a guideline throughout this report. Consequently, I will present the top-most layer of the integration architecture here and defer the discussion of the remaining parts to the following sections. Following

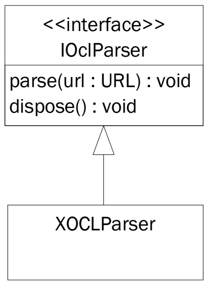
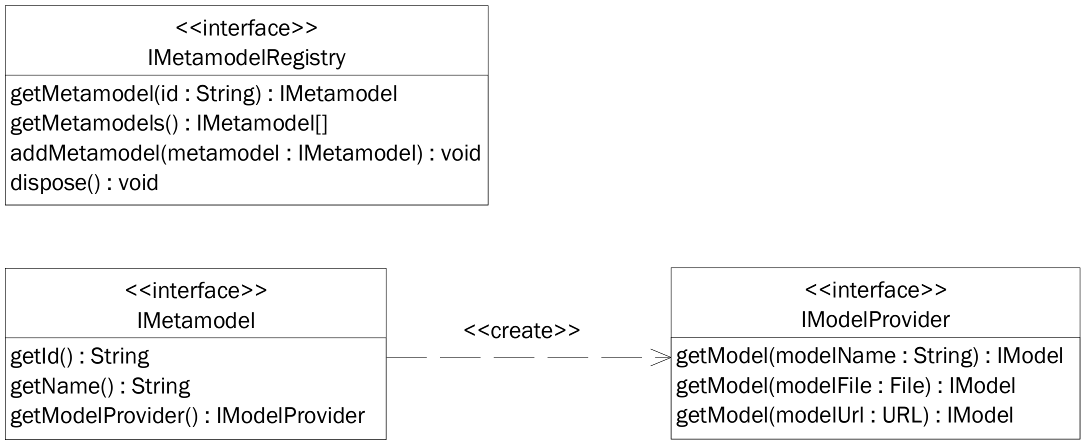


Figure 6.27: *The IOclParser interface realized by the XOCL parser*

the general principle of the entire project, the functionality of the Model Bus is captured in a set of interfaces to promote flexibility and guarantee loose coupling of components. Abstract default implementations for many of these interfaces exist, but for the purpose of brevity, I will omit them in the relevant diagrams.

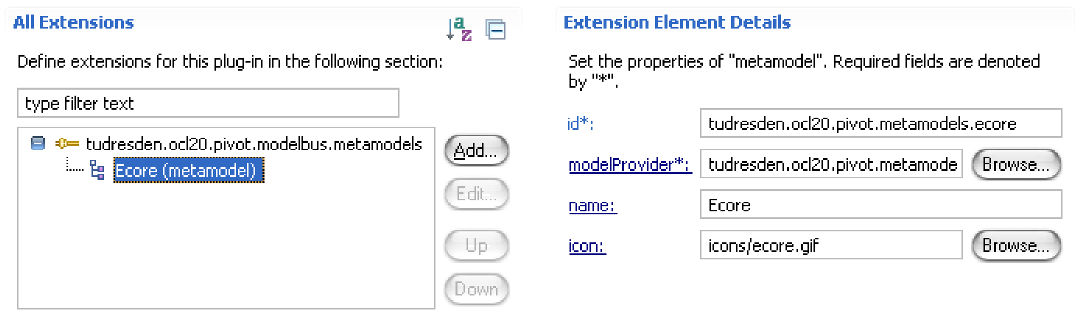
On the *Concepts Level*, the Model Bus defines an abstraction `IMetamodel` representing the metamodels of domain-specific languages that are to be integrated with OCL. The main task of an `IMetamodel` implementation is to grant access to an `IModelProvider` which in turn allows to generically load instances of the DSL in the form of an `IModel`. Finally, an `IMetamodelRegistry` manages all registered `IMetamodel` instances. The design is summarized in Figure 6.28.



<<interface>>
IMetamodelRegistry
getMetamodel(id : String) : IMetamodel
getMetamodels() : IMetamodel[]
addMetamodel(metamodel : IMetamodel) : void
dispose() : void
<<interface>>
IMetamodel
getId() : String
getName() : String
getModelProvider() : IModelProvider
<<create>>
<<interface>>
IModelProvider
getModel(modelName : String) : IModel
getModel(modelFile : File) : IModel
getModel(modelUrl : URL) : IModel

Figure 6.28: *The Model Bus infrastructure on the Concepts Level*

For an integration within Eclipse, I have implemented both the `IMetamodelRegistry` and the `IMetamodel` interface based on a custom extension point. This permits configuring new DSLs declaratively without the need for any further coding (Figure 6.29). In addition, I have developed an `IModelProvider` implementation that is capable of loading Ecore models serialized to XMI.


All Extensions

Define extensions for this plug-in in the following section:

type filter text

tudresden.ocl20.pivot.modelbus.metamodels
Ecore (metamodel)

Add... Edit... Up Down

Extension Element Details

Set the properties of "metamodel". Required fields are denoted by "*".

id*: tudresden.ocl20.pivot.metamodels.ecore

modelProvider*: tudresden.ocl20.pivot.metamode Browse...

name: Ecore

icon: icons/ecore.gif Browse...

Figure 6.29: *Declaratively specifying the integration of a new DSL*

6.2.2 The Definition Level

Implementing model adaptation

The implementation of the *Definition Level* follows the general adapter concept explained in Section 6.1.2. However, to provide a framework for easily creating a new adapter layer for a particular DSL, the actual code is a bit more intricate. Figure 6.30 exemplifies the design once again for the adaptation of the Ecore `EClass` element. For brevity reasons, not all attributes and operations are shown.

As can be seen, the Pivot Model `Type` concept is implemented by a corresponding `TypeImpl` class which is also created by the EMF code generator. The important point to notice is that for each operation from the `Type` interface, I have added a protected operation with the same name, but postfixed with `Gen`. This causes the EMF code generator to forward the code for these methods to their `Gen` counterpart. Listing 6.5 contains an excerpt illustrating the mechanism.

```
/**
 * @generated NOT
 */
public List<Property> getOwnedProperty() {
  return getOwnedPropertyGen();
}

/**
 * @generated
 */
protected final List<Property> getOwnedPropertyGen() {
  if (ownedProperty == null) {
    ownedProperty = new EObjectContainmentWithInverseEList<Property>(
        Property.class,this,PivotModelPackageImpl.TYPE__OWNED_PROPERTY,
        PivotModelPackageImpl.PROPERTY__OWNING_TYPE);
  }
  return ownedProperty;
}
```

Listing 6.5: *Forwarding the code generated by EMF to another method*

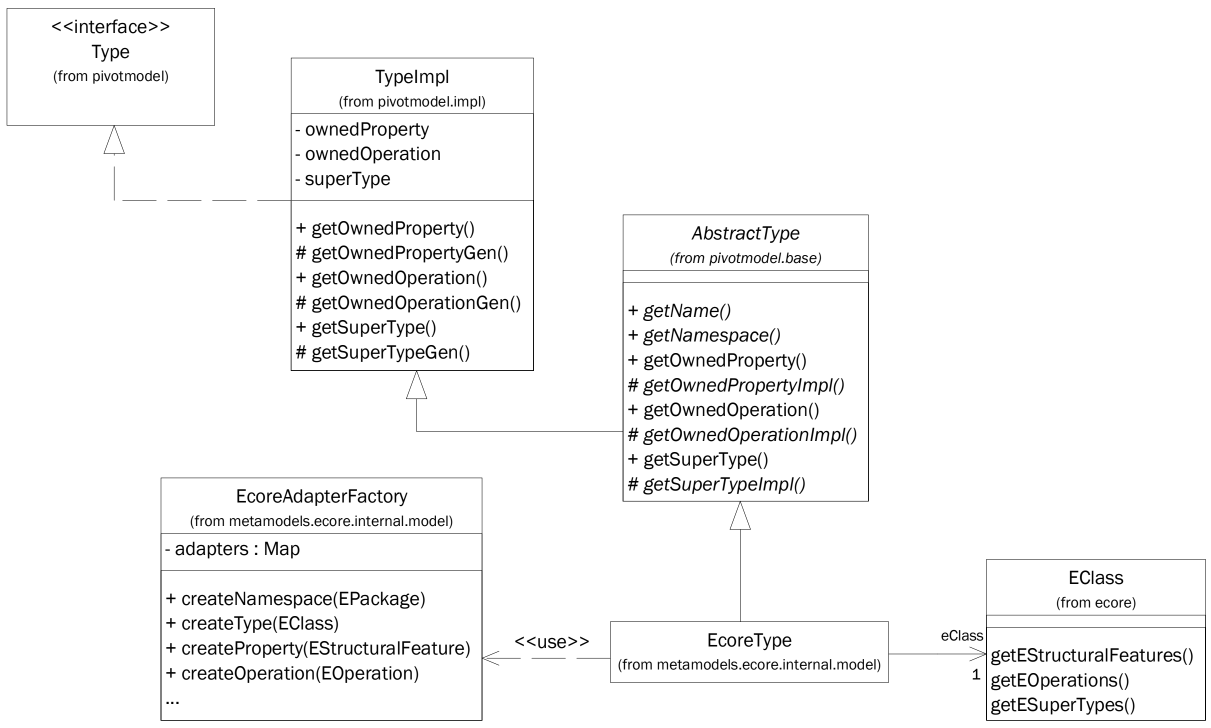


Figure 6.30: *Implementing the Model Adaptation approach*

Now, to understand the motivation behind this seemingly meaningless idiom, recall that the adapters are supposed to store transient elements added to the model. Rather than implementing this mechanism entirely by hand, I have aimed for reusing as much of the EMF implementation as possible. Listing 6.6 shows how the `Gen` methods are used when additional (transient) elements are added to the model.

```java
public Type addProperty(Property property) {
  getOwnedPropertyGen().add(property);
}
```

Listing 6.6: *Adding transient elements to a model*

Through this design, domain-specific adapter subclasses may override the operations declared in the `Type` interface (e.g., `getOwnedProperty`) without losing the implementation created by EMF. This also constitutes a prerequisite for reusing the EMF UI support since it depends on the reflective capabilities additionally generated into the `Impl` classes. Obviously, it is also possible to implement the Pivot Model interfaces from scratch and provide a custom user interface if the dependency to EMF poses a problem. As long as the Pivot Model semantics are realized properly, clients of the interface will remain unaffected.

Returning back to Figure 6.30, the `AbstractType` class still demands an explanation. There is one such abstract base class for each concept in the Pivot Model. This serves two purposes. Firstly, all operations that have to be re-implemented by domain-specific adapter subclasses are overridden and made abstract. Secondly, the last missing piece to support transient model elements is provided. Listing 6.7 shows how an application of the *Template Method* design pattern [GHJV95] ensures the union of transient elements and domain-specific adapters for the `ownedProperty` association.

```java
@Override
public final List<Property> getOwnedProperty() {
  List<Property> ownedProperty = new ArrayList<Property>();
  ownedProperty.addAll(getOwnedPropertyGen());
  ownedProperty.addAll(getOwnedPropertyImpl());
  return ownedProperty;
}

protected abstract List<Property> getOwnedPropertyImpl();
```

Listing 6.7: *Combining transient elements and model adapters*

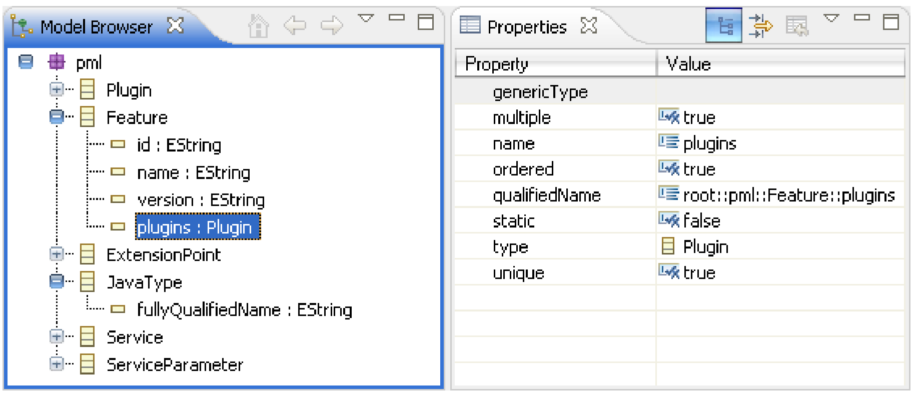
Finally, it is up to the domain-specific adapters to actually implement the concrete model adaptation logic. For this project, I have realized a complete adapter layer for the Ecore metamodel. The creation of new adapters is here delegated to a factory (`EcoreAdapterFactory`) that caches previously created adapters (Listing 6.8).

```java
@Override
protected List<Property> getOwnedPropertyImpl() {
  List<Property> ownedProperty = new ArrayList<Property>();

  for (EStructuralFeature sf : eClass.getEStructuralFeatures()) {
    ownedProperty.add(EcoreAdapterFactory.INSTANCE.createProperty(sf));
  }

  return ownedProperty;
}
```

Listing 6.8: *Implementing the adapter to the Ecore model*

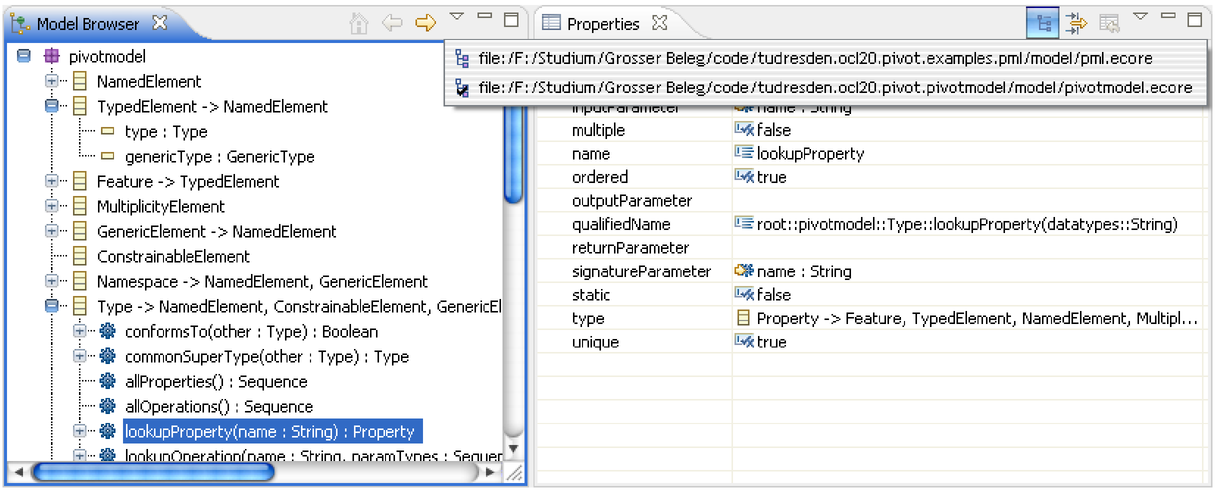


```
pml
  Plugin
  Feature
    id : EString
    name : EString
    version : EString
    plugins : Plugin
  ExtensionPoint
  JavaType
    fullyQualifiedName : EString
  Service
  ServiceParameter
```

Property	Value
genericType	
multiple	true
name	plugins
ordered	true
qualifiedName	root::pml::Feature::plugins
static	false
type	Plugin
unique	true

Figure 6.31: *The PML metamodel in the Model Browser*

Once the adapters for a target DSL have been written, instances of that DSL (i.e., models) can be accessed via the Pivot Model interfaces. In particular, they can be visualized using a single user interface. Figures 6.31 and 6.32 show the metamodels of PML and the Pivot Model itself in the DSL-agnostic model browser developed in the course of this project. Notice how the Properties View displays the property values of the corresponding Pivot Model metaclasses. For instance, the value of the property '`isMultiple`' is determined based on the upper bound of the adapted `ETypedElement`.



```
pivotmodel
  NamedElement
  TypedElement -> NamedElement
    type : Type
    genericType : GenericType
  Feature -> TypedElement
  MultiplicityElement
  GenericElement -> NamedElement
  ConstrainableElement
  Namespace -> NamedElement, GenericElement
  Type -> NamedElement, ConstrainableElement, GenericEl
    conformsTo(other : Type) : Boolean
    commonSuperType(other : Type) : Type
    allProperties() : Sequence
    allOperations() : Sequence
    lookupProperty(name : String) : Property
    lookupOperation(name : String, paramTypes : Sequer
```

```
file:/F:/Studium/Grosser Beleg/code/tudresden.ocl20.pivot.examples.pml/model/pml.ecore
file:/F:/Studium/Grosser Beleg/code/tudresden.ocl20.pivot.pivotmodel/model/pivotmodel.ecore
```

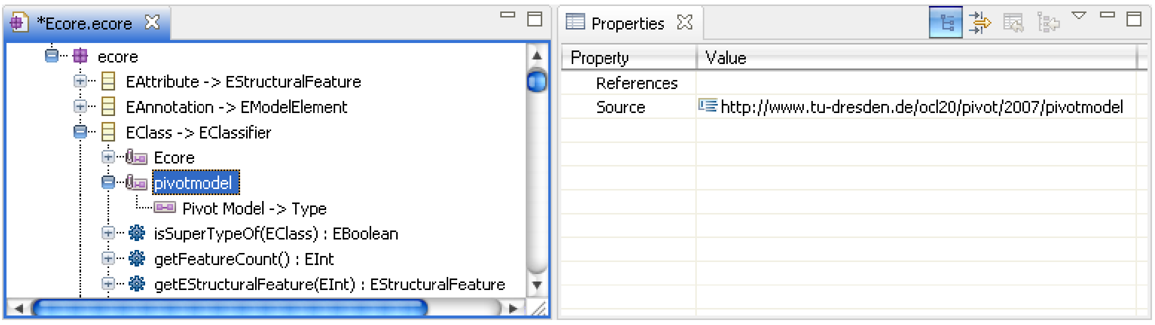
inputParameter	name : String
multiple	false
name	lookupProperty
ordered	true
outputParameter	
qualifiedName	root::pivotmodel::Type::lookupProperty(datatypes::String)
returnParameter	
signatureParameter	name : String
static	false
type	Property -> Feature, TypedElement, NamedElement, Multipl...
unique	true

Figure 6.32: *The metamodel of the Pivot Model adapted by itself*

Automatically generating the adapters for a target DSL

The manual implementation of the adapter layer for the Ecore metamodel revealed that the actual adaptation logic represents only a small fraction of the code. This gives rise to the idea of generating most of the surrounding boilerplate code automatically. As outlined in Section 6.1.2, this basically requires a mechanism to declaratively specify the mapping of Pivot Model concepts to the corresponding elements in the target DSL. Following is a brief description of how to realize this in the Eclipse Modeling Framework. Unfortunately, limited time did not allow me to actually implement the approach.

The core idea is to annotate the model of the target DSL and use this information to generate appropriate adapter skeletons. Figure 6.33 illustrates a suitable annotation of the Ecore EClass element. To adapt the EMF code generator, a custom `GeneratorAdapterFactory` is registered in an extension of the `org.eclipse.emf.codegen.ecore.generatorAdapters` extension point. This factory in turn creates the `GeneratorAdapter` instances that perform the actual code generation. EMF provides its own template engine called *JET (Java Emitter Templates)* which allows to create arbitrary textual content in a syntax resembling that of *Java Server Pages (JSP)*. A JET template is parameterized with elements of a *generator model* that wrap the objects from the target DSL. Thus, the template may easily access the annotations and generate the corresponding adapter code.



```
ecore
  EAttribute -> EStructuralFeature
  EAnnotation -> EModelElement
  EClass -> EClassifier
    Ecore
    pivotmodel
      Pivot Model -> Type
    isSuperTypeOf(EClass) : EBoolean
    getFeatureCount() : EInt
    getEStructuralFeature(EInt) : EStructuralFeature
```

Property	Value
References	
Source	http://www.tu-dresden.de/ocl20/pivot/2007/pivotmodel

Figure 6.33: *Annotating a target DSL to define the Pivot Model mapping*

The following items need to be generated for a target DSL:

- an adapter for each annotated element
- an adapter factory class containing a `create` method for each Pivot Model concept that has an annotated equivalent in the target DSL

Integrating the model of the OCL Standard Library

A particularly challenging task in the development of an OCL engine is the integration of the OCL Standard Library on the *Definition Level*. An OCL parser should be able to find the following elements when building the abstract syntax tree:

- operations defined for `OclAny`, the implicit super type of all model types
- the predefined OCL primitive types and their operations
- any of the other predefined types (`OclVoid`, `OclInvalid`, `OclType`)
- user-defined collection and tuple types

Existing implementations, such as the Dresden OCL2 Toolkit and the Kent OCL Library, usually create the model of the OCL Standard Library programmatically. This stems from the fact that there is an infinite number of tuple and collection types which have to be dynamically created when parsing an expression. Moreover, some of the predefined operations in the Library have return types that depend on the object they are invoked on. Examples are `OclAny::asSet` (returning a singleton set containing the object) and `OclAny::allInstances` (returning the set of all instances of a type). Both operations have `Set(T)` as their return type, but the concrete binding for `T` cannot be determined until the source type of the operation call is known.

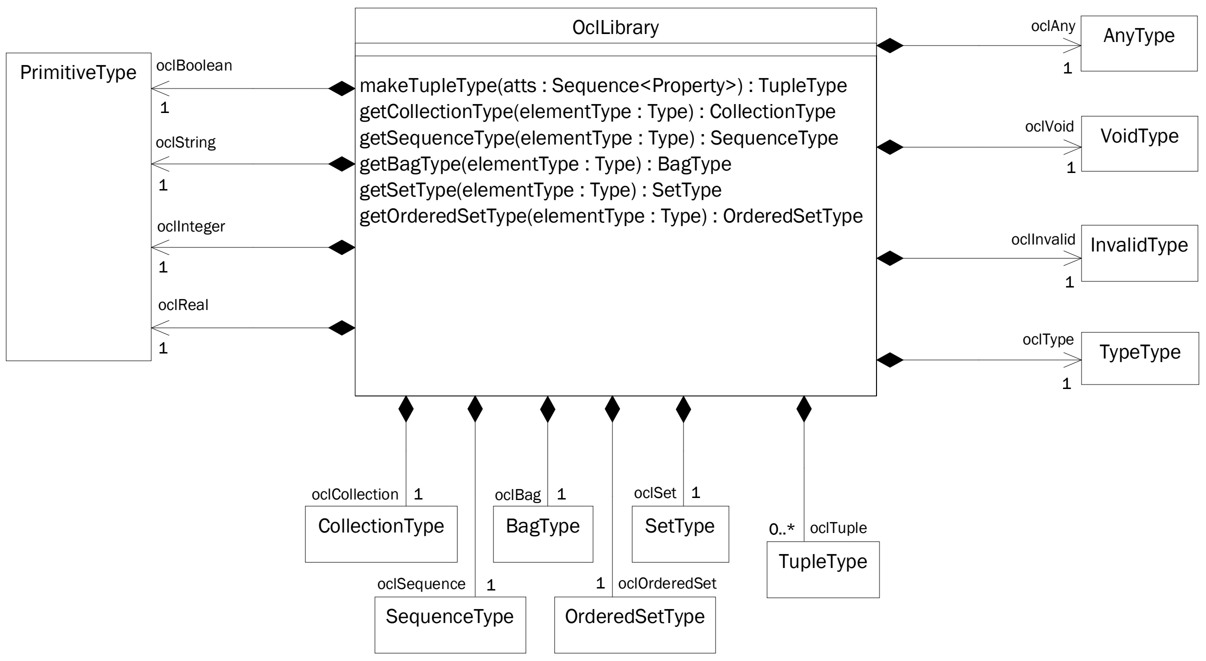

OclLibrary
makeTupleType(atts : Sequence<Property>) : TupleType
getCollectionType(elementType : Type) : CollectionType
getSequenceType(elementType : Type) : SequenceType
getBagType(elementType : Type) : BagType
getSetType(elementType : Type) : SetType
getOrderedSetType(elementType : Type) : OrderedSetType
PrimitiveType
oclBoolean
oclString
oclInteger
oclReal
1
oclAny
AnyType
oclVoid
VoidType
oclInvalid
InvalidType
oclType
TypeType
oclCollection
CollectionType
oclBag
BagType
oclSet
SetType
0..*
oclTuple
TupleType
oclSequence
SequenceType
oclOrderedSet
OrderedSetType

Figure 6.34: *The OclLibrary facade class providing access to the model of the Standard Library*

In this work, I propose the novel approach of modeling the OCL Standard Library externally as an instance of the Pivot Model. This has a number of significant advantages. In particular, the structure of the Library can easily be validated, altered, extended and modularized because it is no longer hidden inside the code. Moreover, the complexity of the code reduces which benefits testability and maintainability. In Section 6.1.1, I have already described in detail how the Generics support in the Pivot Model provides the necessary conceptual foundations. Therefore, I will limit the discussion here to a number of implementation-related issues.

An important question is how the model of the Standard Library can be accessed after it has been loaded from its XMI serialization. This is mainly required when determining the type of an OCL expression since the result may be one of the OCL predefined types. Recall from the previous section that type evaluation is performed directly in the expression classes by suitably overriding the `getType` method. To provide a single point of entry for retrieving the Standard Library types, I have defined a facade class OclLibrary that offers the required functionality (Figure 6.34). A similar pattern is employed by the current Dresden OCL2 Toolkit. In contrast to the Toolkit implementation, however, I have not realized the OclLibrary facade as a *Singleton*. Instead, it is modeled as an element of the Essential OCL Types package resulting in a corresponding interface `OclLibrary` and implementation class `OclLibraryImpl` generated by EMF. This further ensures a purely interface-based API without explicit dependencies to implementation classes. To allow access to the OclLibrary, the design introduces an association from the OclExpression meta class (Figure 6.35). Following the *Dependency Injection* design pattern [Fow04], this reference needs to be initialized when building the abstract syntax tree of an OCL expression. Further below is a more detailed description how this is facilitated.

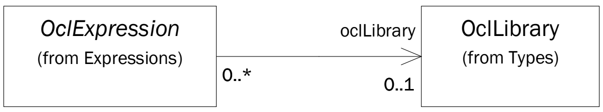


Figure 6.35: *Each OCL expression holds a reference to the OCL Library*

Now that each expression class has convenient access to the Standard Library, realizing the type evaluation logic is straightforward. Listing 6.9 exemplifies the principle by showing the `getType` implementation inside `PropertyCallExpImpl`.

```
1  @Override
2  public Type getType() {
3    if (referredProperty == null) {
4      throw new WellformednessException(...);
5    }
6    return getOclType(referredProperty.getType());
7  }
```

Listing 6.9: *Type evaluation for property call expressions*

Notice the call to `getOclType` in line 6. This method is implemented in the super class `OclExpressionImpl` (see Listing 6.10) and serves two purposes. Firstly, it maps primitive model types based on their `PrimitiveTypeKind` (see Section 6.1.1) to the corresponding OCL type from the Standard Library. Secondly, it ensures that all types except the OCL collection types descend from `OclAny` thereby inheriting all of its predefined operations. Once again, I follow an "on demand" pattern here, rather than establishing the inheritance relationship with `OclAny` for all model elements immediately after parsing (as implemented in the current Toolkit). This should yield significant performance gains for larger models.

```java
protected Type getOclType(Type type) {
  if (oclLibrary == null) {
    throw new IllegalStateException(...);
  }

  if (type instanceof PrimitiveType) {
    type = mapPrimitiveType((PrimitiveType) type);
  }
  else if (!(type instanceof CollectionType)) {
    type = ensureDescendanceFromOclAny(type);
  }

  return type;
}
```

Listing 6.10: *Ensuring OCL semantics for model types*

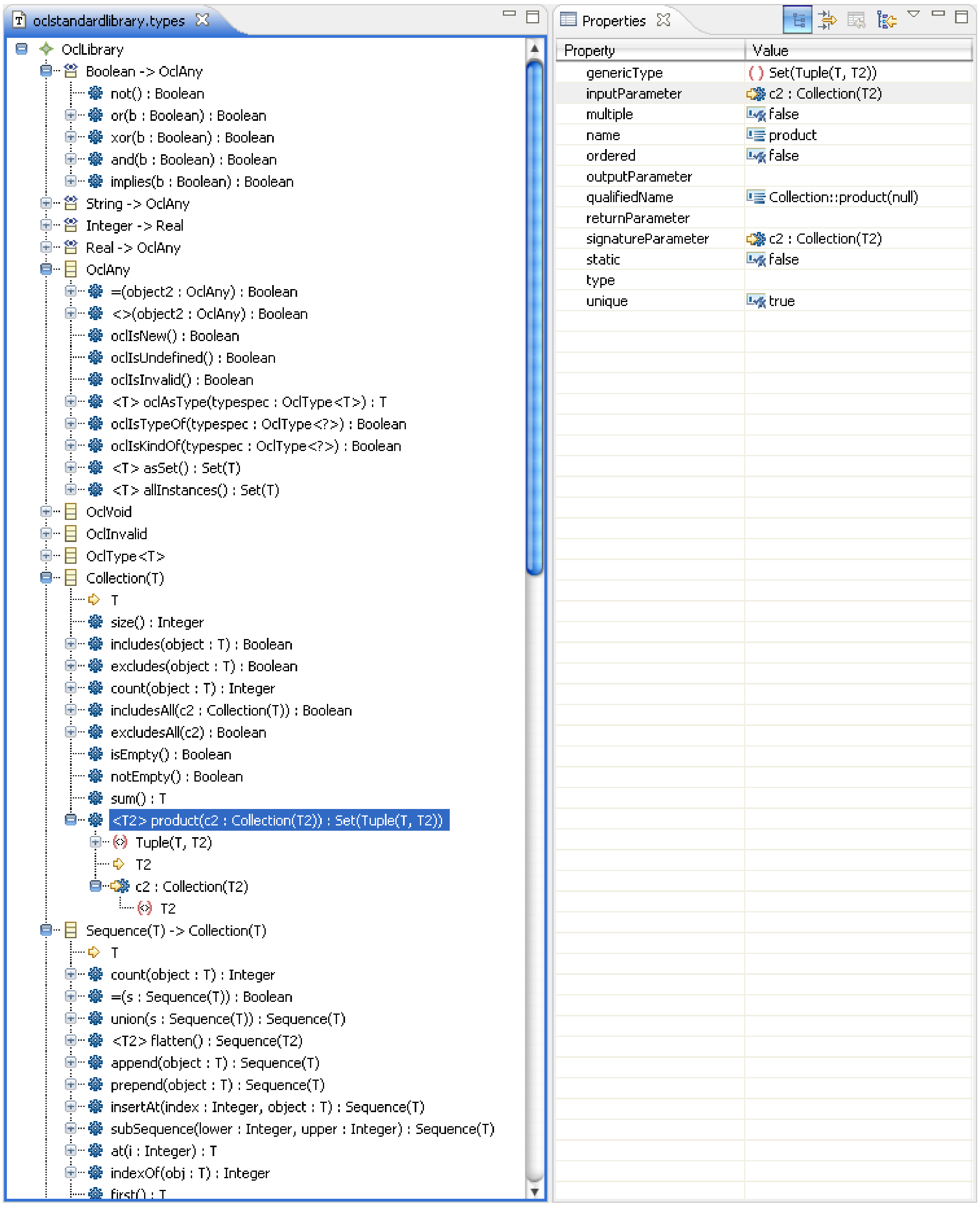
A final question that arises from the remarks above is how to integrate the predefined types `OclVoid` and `OclInvalid` with types from the model. The OCL Specification states that both conform to all other types [OMG06c, p. 138]. As an answer, the current Dresden OCL2 Toolkit implementation introduces inheritance relationships to all model types which do not have subtypes. When adapting foreign model repositories via the ModelFacade mechanism (see Section 5.1.3), these relationships are dynamically created *each* time when the parents of the `OclVoid` type are requested (the Toolkit does not implement the `OclInvalid` type). For large models, this represents a tremendous performance bottleneck since finding all types without subtypes might be non-trivial and expensive. In the Pivot Model implementation, I have opted for another solution (Listing 6.11). By simply overriding conformance-related methods in `VoidTypeImpl` and `InvalidTypeImpl`, respectively, the correct semantics can be provided without introducing elaborate inheritance relationships.

```java
@Override
public boolean conformsTo(Type other) {
  return true;
}

@Override
public Type commonSuperType(Type other) {
  return other;
}
```

Listing 6.11: *Ensuring type conformance of OclVoid and OclInvalid with types from the model*

Based on the design illustrated above, it is now very easy to create the model of the OCL Standard Library. Figure 6.36 shows an excerpt as displayed in the (heavily customized) Pivot Model editor generated by EMF. Notice how `Tuple(T, T2)` is defined as the type argument for the generic return type of the `Collection::product` operation. The root of the model is the `OclLibrary` facade element. Once all of its reference slots have been filled, the model of the Standard Library is complete and guaranteed to work within the OCL engine.



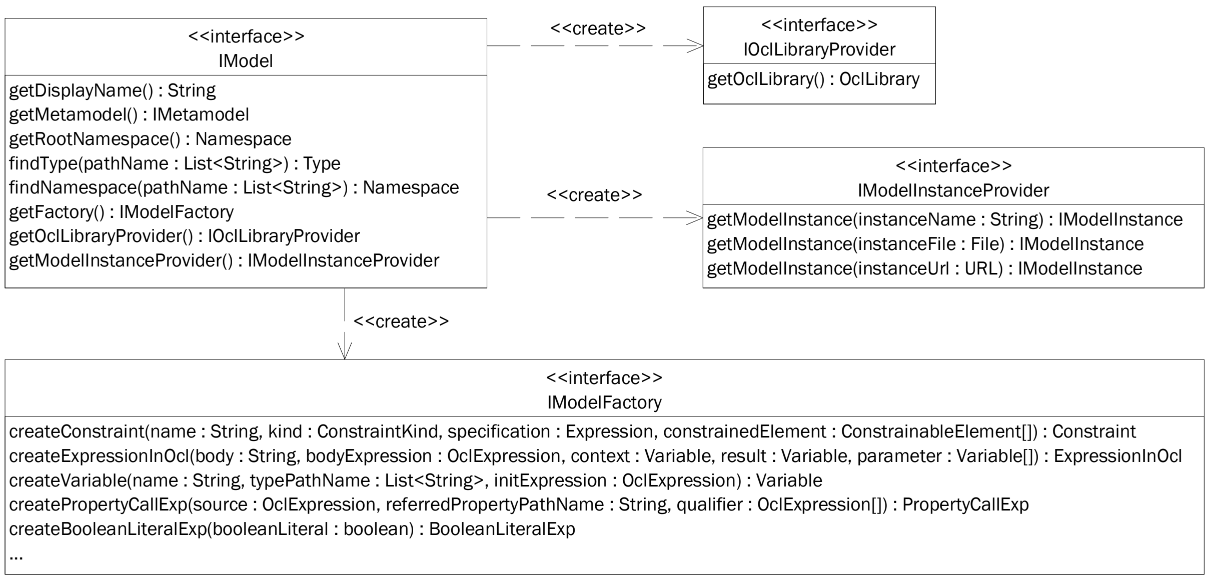
```
OclLibrary
  Boolean -> OclAny
    not() : Boolean
    or(b : Boolean) : Boolean
    xor(b : Boolean) : Boolean
    and(b : Boolean) : Boolean
    implies(b : Boolean) : Boolean
  String -> OclAny
  Integer -> Real
  Real -> OclAny
  OclAny
    =(object2 : OclAny) : Boolean
    <>(object2 : OclAny) : Boolean
    oclIsNew() : Boolean
    oclIsUndefined() : Boolean
    oclIsInvalid() : Boolean
    <T> oclAsType(typespec : OclType<T>) : T
    oclIsTypeOf(typespec : OclType<?>) : Boolean
    oclIsKindOf(typespec : OclType<?>) : Boolean
    <T> asSet() : Set(T)
    <T> allInstances() : Set(T)
  OclVoid
  OclInvalid
  OclType<T>
  Collection(T)
    T
    size() : Integer
    includes(object : T) : Boolean
    excludes(object : T) : Boolean
    count(object : T) : Integer
    includesAll(c2 : Collection(T)) : Boolean
    excludesAll(c2) : Boolean
    isEmpty() : Boolean
    notEmpty() : Boolean
    sum() : T
    <T2> product(c2 : Collection(T2)) : Set(Tuple(T, T2))
      Tuple(T, T2)
      T2
      c2 : Collection(T2)
        T2
  Sequence(T) -> Collection(T)
    T
    count(object : T) : Integer
    =(s : Sequence(T)) : Boolean
    union(s : Sequence(T)) : Sequence(T)
    <T2> flatten() : Sequence(T2)
    append(object : T) : Sequence(T)
    prepend(object : T) : Sequence(T)
    insertAt(index : Integer, object : T) : Sequence(T)
    subSequence(lower : Integer, upper : Integer) : Sequence(T)
    at(i : Integer) : T
    indexOf(obj : T) : Integer
    first() : T
```

Property	Value
genericType	Set(Tuple(T, T2))
inputParameter	c2 : Collection(T2)
multiple	false
name	product
ordered	false
outputParameter	
qualifiedName	Collection::product(null)
returnParameter	
signatureParameter	c2 : Collection(T2)
static	false
type	
unique	true

Figure 6.36: *Modeling the OCL Standard Library*

Extending the integration architecture

On the *Definition Level*, four interfaces are added to the Model Bus architecture introduced in the previous section (see Figure 6.37). Among the noteworthy features are the two `IModel` operations `findType` and `findNamespace` that are declared in the OCL 2.0 Specification [OMG06c, p. 167]. Placing these operations in a designated model abstraction is an advancement over the current Dresden OCL2 Toolkit where they are erroneously located in the `Package` metaclass.



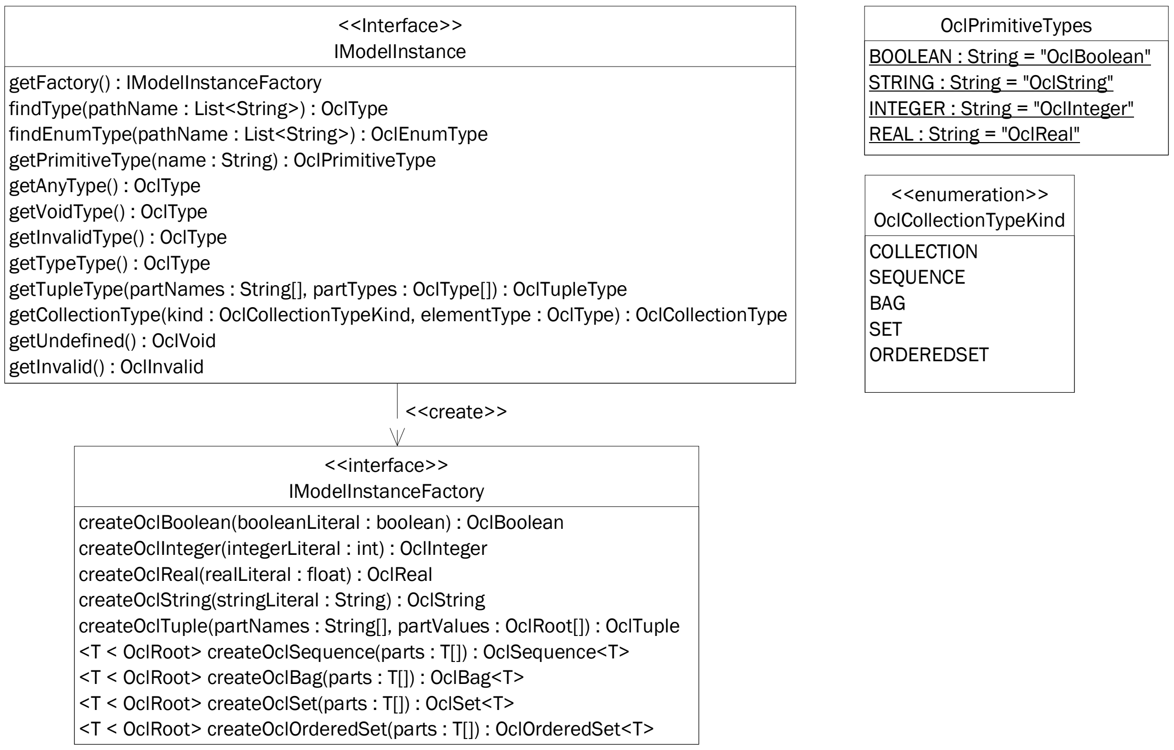
<<interface>>
IModel
getDisplayName() : String
getMetamodel() : IMetamodel
getRootNamespace() : Namespace
findType(pathName : List<String>) : Type
findNamespace(pathName : List<String>) : Namespace
getFactory() : IModelFactory
getOclLibraryProvider() : IOclLibraryProvider
getModelInstanceProvider() : IModelInstanceProvider
<<create>>
<<interface>>
IOclLibraryProvider
getOclLibrary() : OclLibrary
<<create>>
<<interface>>
IModelInstanceProvider
getModelInstance(instanceName : String) : IModelInstance
getModelInstance(instanceFile : File) : IModelInstance
getModelInstance(instanceUrl : URL) : IModelInstance
<<create>>
<<interface>>
IModelFactory
createConstraint(name : String, kind : ConstraintKind, specification : Expression, constrainedElement : ConstrainableElement[]) : Constraint
createExpressionInOcl(body : String, bodyExpression : OclExpression, context : Variable, result : Variable, parameter : Variable[]) : ExpressionInOcl
createVariable(name : String, typePathName : List<String>, initExpression : OclExpression) : Variable
createPropertyCallExp(source : OclExpression, referredPropertyPathName : String, qualifier : OclExpression[]) : PropertyCallExp
createBooleanLiteralExp(booleanLiteral : boolean) : BooleanLiteralExp
...

Figure 6.37: *The Model Bus infrastructure on the Definition Level*

Next, the `IModelFactory` interface represents the central link between an OCL parser and the adapted model. The *AbstractFactory* design pattern [GHJV95] used here allows to generically build the abstract syntax tree for an OCL expression. Notice that a parser can simply pass in strings to denote model elements (e.g., the referred property of a `PropertyCallExp` or the type of a `Variable`) rather than browsing the model itself. A complete implementation of `IModelFactory` is available in the Model Bus Eclipse plug-in. It not only ensures input validation and error handling, but also properly initializes the reference to the `OclLibrary` facade for newly created OCL expressions. To this end, it retrieves the `OclLibrary` from an `IOclLibraryProvider` which in turn is obtained from the associated `IModel`. The default implementation of `IOclLibraryProvider` simply loads the model of the Standard Library from its XMI serialization located in a sub-folder of the Model Bus plug-in. Lastly, the `IModelInstanceProvider` interface provides the connection to the Execution Level.

6.2.3 The Execution Level

As pointed out earlier, the Execution Level is actually outside the scope of this report and has only been included for conceptual completeness. Consequently, within this project I merely provide a corresponding Eclipse plug-in and the interfaces of the improved OCL Standard Library design as suggested in Section 6.1.3. Additionally, the Model Bus architecture described in the previous sections is completed with the elements required for evaluating OCL expressions via an interpreter or code generator (Figure 6.38).



<<Interface>>
IModelInstance
getFactory() : IModelInstanceFactory
findType(pathName : List<String>) : OclType
findEnumType(pathName : List<String>) : OclEnumType
getPrimitiveType(name : String) : OclPrimitiveType
getAnyType() : OclType
getVoidType() : OclType
getInvalidType() : OclType
getTypeType() : OclType
getTupleType(partNames : String[], partTypes : OclType[]) : OclTupleType
getCollectionType(kind : OclCollectionTypeKind, elementType : OclType) : OclCollectionType
getUndefined() : OclVoid
getInvalid() : OclInvalid
<<create>>
<<interface>>
IModelInstanceFactory
createOclBoolean(booleanLiteral : boolean) : OclBoolean
createOclInteger(integerLiteral : int) : OclInteger
createOclReal(realLiteral : float) : OclReal
createOclString(stringLiteral : String) : OclString
createOclTuple(partNames : String[], partValues : OclRoot[]) : OclTuple
<T < OclRoot> createOclSequence(parts : T[]) : OclSequence<T>
<T < OclRoot> createOclBag(parts : T[]) : OclBag<T>
<T < OclRoot> createOclSet(parts : T[]) : OclSet<T>
<T < OclRoot> createOclOrderedSet(parts : T[]) : OclOrderedSet<T>
OclPrimitiveTypes
BOOLEAN : String = "OclBoolean"
STRING : String = "OclString"
INTEGER : String = "OclInteger"
REAL : String = "OclReal"
<<enumeration>>
OclCollectionTypeKind
COLLECTION
SEQUENCE
BAG
SET
ORDEREDSET

Figure 6.38: *The Model Bus infrastructure on the Execution Level*

At the core of the design is the `IModelInstance` interface that enables access to the meta level of a domain-specific execution environment. Thus, it allows to find the type referenced by a `TypeLiteralExp` contained in the abstract syntax tree of an OCL expression. The operations `getUndefined` and `getInvalid` return the domain-specific representations of the single instances of `OclVoid` and `OclInvalid`, respectively. Conversely, an `IModelInstanceFactory` allows creating new adapters for domain-specific primitive types, collection types, and tuple types to accommodate the corresponding variants of `LiteralExp`. Finally, two additional mixin interfaces provide the foundations for a future development of a QVT engine (Figure 6.39). In analogy to the design presented in Section 6.1.3, the prefix `Mutable` symbolizes the additional capability of creating instances of arbitrary model elements in the target model during a model transformation.

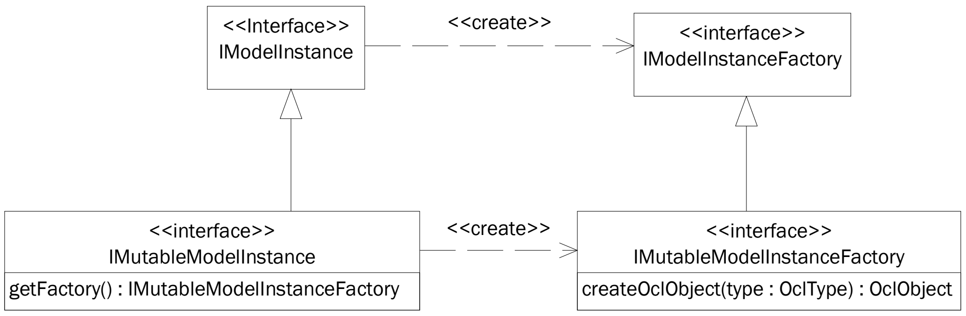


Figure 6.39: *An extension of the Model Bus infrastructure to support a QVT engine*

The `IModelInstance` interface provides the solution to the last open question from Section 5.1.4. Recall that a fundamental drawback of the current Dresden OCL2 Toolkit implementation is the explicit dependency of generated code on a particular model repository technology. In the interface-based Model Bus architecture, these implementation details are abstracted away which greatly enhances reusability and also paves the way for runtime evaluation via an interpreter. Finally, the `IMutableModelInstanceFactory` interface provides the necessary means to address the *system instantiation* problem identified in Section 2.1.3. By hiding the necessary semantics in a domain-specific implementation of the Execution Level, a model transformation engine can generically instantiate system-level elements and initialize their properties. As outlined in Section 2.2.2, this opens up the possibility for metamodel-based data conversion using a standardized transformation language like QVT.

7 Discussion

In this chapter, I will review the results achieved during this project and assess their benefits and possible shortcomings. Then, I will clearly highlight the contributions made by this work to prove its significance. Finally, I will conclude the report and give some directions for further research.

7.1 Evaluation

Unsurprisingly, implementing the Pivot Model adapters for Ecore did not pose any problems, given the relative similarity of its metamodel. Thus, this section is not going to discuss the "completeness" of the Pivot Model, but rather elaborate on how well the infrastructure developed around it supports the integration of a new target language. In other words, I am not evaluating the *effectivity* but the *efficiency* of the solution presented in this report. Further research is needed to investigate how well the Pivot Model abstraction supports a breadth of different DSLs and to what extent OCL lends itself to being used as a constraint and query mechanism for these languages.

In terms of actual coding volume, the Ecore integration required little work. Even without the code generation support described in Section 6.2.2, tedious and repetitive programming was reduced to a minimum through the code assistance in the Eclipse IDE. For instance, skeletons for the abstract methods in the adapter base classes were filled in automatically. Table 7.1 compares the lines of code required for adapting the metamodels of a number of target languages in the Dresden OCL2 Toolkit[1], the Kent OCL Library and the Pivot Model architecture. Admittedly, this represents a rather imprecise measure for comparison, but the numbers still give an indication for assessing the benefit yielded by the results of this work. Adding the suggested code generation facilities will further reduce the effort for integrating OCL with domain-specific languages. It should be noted, though, that the current Ecore adapter layer represents the absolute minimum required for a fully functional integration. To improve performance, further methods of the default implementation may be overridden. Prime candidates are the various `lookup` operations that often have equivalent counterparts in the Ecore model classes.

	Dresden OCL2 Toolkit		**Kent OCL**	**Pivot Model**
Adapted metamodel	UML	MOF	Ecore	Ecore
Lines of Code	2124	1657	685	554

Table 7.1: *Comparison of effort required for adapting a DSL*

As described in Section 6.2.2, the generic capabilities of the Pivot Model allowed for a novel approach to integrate the OCL Standard Library. After the necessary tool support had been developed, creating the model of the Library proved to be intuitive and simple. To the best of my knowledge, there is no published work that can match the flexibility and extensibility of my solution. Of course, its main advantages will only really pay off in the light of future extension

[1] not including code required for a custom `ModelFacade` to adapt foreign model repositories

or modularization of the OCL Standard Library. As of now, the main contribution of my work is to reduce the inherent complexity of the OCL implementation code. In the past, this has been a major impediment to the productivity of thesis students when doing research around the Dresden OCL2 Toolkit.

As a side note, the comprehensive UI support developed for this project also greatly assisted the implementation of the Ecore adapters. By simply browsing through test models visually, the correctness of the code could easily be validated without the need for elaborate test cases. Certainly, dedicated tests must not be omitted in a professional setting, but for prototypical explorations and “proof-of-concept” studies they are usually dispensable. Development scenarios employing agile methodologies to investigate a great variety of domain-specific languages before committing to a fixed modeling toolset may particularly benefit from this approach.

7.2 Limitations

Due to the prototypical nature of the implementation, some limitations exist. For instance, the XOCL parser does not yet fully parse the context declaration for constraints over operations and properties. So far, only the required variables (`self`, `result` and parameter variables) are created, but the code for locating the constrained elements in the model is still missing. This can easily be added but since the XOCL parser is destined to be replaced soon anyways, it might not become a necessity.

Also, there still remains an unaddressed problem regarding the type evaluation for operation call expressions. Recall from Section 6.1.1 that generic operations may not have a valid type until their type parameters are bound. The concrete types to be used for the binding depend (1) on the arguments of the operation (as in `Collection::product` or `OclAny::oclAsType`) or (2) the source type of the operation call (as in `OclAny::asSet` or `OclAny::allInstances`). Currently, `OperationCallExp::getType` simply treats these cases explicitly by branching on the name of the referred operation. The implementation therefore “knows” how to determine the correct binding for a particular operation in the OCL Standard Library. This dedicated handling of predefined operations is similar to the existing implementation in the Dresden OCL2 Toolkit, yielding the same significant conceptual flaw of introducing details of the Standard Library (M1) into the implementation of the OCL abstract syntax elements (M2).

An obvious solution applicable to scenario (1), where the binding depends on the actual arguments passed to a generic operation, is to imitate the behavior of the Java 5 compiler. That requires determining which types are bound to those slots where the operation signature defines its generic type parameters. Due to infinite nesting of generic types, however, an implementation of this approach may be nontrivial and, thus, has not been realized within the time frame available for this report. Also, I have not yet found a satisfactory solution for scenario (2), so further efforts are required to achieve a truly clean design where no dependencies to the Standard Library remain on meta layer M2.

Lastly, I want to highlight that a limitation “by design” is the lacking support for UML behavioral concepts such as states, signals and actions. Obviously, these UML-specific meta elements would severely hamper the domain independence of the Pivot Model. However, validating UML state machines or other behavioral models now requires an explicit mapping of the Pivot Model concepts to the corresponding UML meta classes. Further research should evaluate the appropriateness of the Pivot Model abstraction for this purpose.

7.3 Contributions of this Work

In this report, I have thoroughly analyzed existing ideas that have been proposed to integrate OCL with different metamodels. Based on these considerations, I have developed a new design for a DSL-agnostic OCL engine. The work was done against the background of ongoing extension and evolution of the Dresden OCL2 Toolkit, but the results and insights gained may be beneficial within a broader scope as well. This section will briefly list the main contributions of this project, both in regard to continuing research at Dresden Unversity of Technology and the advancement of model-driven approaches in general. In the following two sections, I will provide a more detailed summary of the report and give directions for future work.

To sum up, in this work I have:

- provided a detailed analysis of inherent conceptual challenges arising from model-driven development scenarios involving (several) domain-specific modeling languages
- suggested a three-layered conceptual framework to ease the classification and evaluation of approaches to integrating OCL with different metamodels
- coherently described and evaluated key design concepts of the current Dresden OCL2 Toolkit which had previously been largely undocumented; this should assist future thesis students in familiarizing themselves with the complex implementation more quickly
- presented a pivotal metamodel for the adaptation of arbitrary DSLs that has been carefully designed for maximum expressive power of OCL queries defined over its instances
- proposed a novel approach to integrating the OCL Standard Library by introducing template types and operations on the model level
- realized a clean and purely interface-based integration architecture that may serve as a foundation for future developments

7.4 Summary and Conclusions

The aim of this report was to design a pivotal metamodel in order to allow OCL queries to be evaluated over instances of arbitrary domain-specific languages. In addition, a flexible and extensible mechanism was required to define the corresponding mapping. In conclusion, I have achieved these goals to a large extent. The design of the Pivot Model presented in this report clearly surpasses previous solutions in terms of conceptual clarity and expressiveness. The model adaptation mechanism, which facilitates the composition of an arbitrary DSL with the Pivot Model, has proved to be effective and efficient, although the prototypical implementation is still lacking the recommended code generation features. A thorough review of the literature and a careful analysis of previous approaches support the arguments made in this report. In this section, I will briefly summarize the most important aspects of this work and draw some final conclusions.

In the beginning, I singled out challenges for the integration of arbitrary DSLs with a standardized query and constraint language such as OCL. These were the ontological classification problem (requiring an explicit mapping of domain concepts to the linguistic elements expected by OCL) and the system instantiation problem (demanding an abstraction from the system-level element instantiation semantics).

Then, I examined existing approaches to model composition within the bounds of a self-defined conceptual framework. I concluded that an adapter-based mechanism is most suitable in the context of this report and preferable over other methods such as model merge. Furthermore, I conducted a detailed requirements analysis based on a simple example DSL.

Thereafter, I presented a comprehensive analysis of existing implementations that try to address the same or similar problems as this report. In my discussion, I highlighted the respective strengths and weaknesses of the Dresden OCL2 Toolkit, the Kent OCL Library and the Epsilon Platform. I paid particular attention to the Dresden OCL2 Toolkit implementation since it forms the background of my own work.

I continued by thoroughly describing the structure of the Pivot Model and the rationale behind some of the design decisions leading to its final form. I explained how the EMOF metamodel was extended to fully leverage the expressive power of the Object Constraint Language. Also, I justified some minor changes made to the Essential OCL metamodel in an attempt to correct apparent discrepancies and insufficiencies in the specification. I showed that the model adaptation principle is powerful enough to integrate structurally heterogeneous DSLs residing in any form of model repository accessible via a Java API. Finally, I explained some of the more intricate details of the EMF-based prototypical implementation. I highlighted how variability and extensibility can be ensured through an overall design scheme that completely relies on well-defined interfaces as the only form of component connection.

Summing up, this report has shown that an integration of OCL with arbitrary domain-specific languages is possible, but requires careful consideration of a number of different issues. The definition of a common metamodel, the selection of a model composition mechanism as well as the design of an integration architecture all raise non-trivial questions. Many more challenges have to be met in order to achieve the long-term goal of a more industrialized and automatized software development industry as pictured in Section 1.1. Since the paradigm of multi-domain modeling using a variety of different DSLs continues to receive attention in the MDSD community, the Pivot Model principle may represent an important step towards more correct domain-specific models and, ultimately, more precision in modeling complex software systems.

7.5 Future Work

Naturally, a single report cannot exhaustively cover the complex topic that forms the background of this project. Together with [Wen06b], this work represents one of the first attempts to extend the scope of the Dresden OCL2 Toolkit beyond an OCL engine for MOF and UML models. Since I have re-engineered and rewritten most of the original implementation in order to modernize the outdated MDR-based infrastructure, the available time did not suffice to explore more of the practical challenges involved with integrating arbitrary DSLs into an OCL engine. This leaves room for exciting future research which hopefully will have a good foundation in the infrastructure components developed for this report.

Of the possible topics for continuing thesis projects, two have already been fixed and committed to. One project will refactor the existing OCL 2.0 parser to integrate it with the new Model Bus infrastructure and thus eliminate the need for the prototypical XOCL language used so far. Another work aims to develop an OCL interpreter operating on the EMF-based Essential OCL implementation created for this report. This motivates the considerable effort I have devoted to identifying potential pitfalls and design flaws in the current Toolkit which would jeopardize the implementation of such an interpreter. In parallel, Pivot Model bindings for UML 1.5 and MOF 1.4 are supposed to be added.

Medium- and long-term goals in relation to a further evolution of the Dresden OCL2 Toolkit are to port the existing tools for code generation [Hei06, Bra06], model transformation [Wen06a] and abstract syntax model visualization [Gär06] to the new platform. Eventually, a metamodel-agnostic QVT engine may provide the foundations for original research into new application domains for model transformations.

Furthermore, the limitations of this report's prototypical implementation can serve as a guideline for future work. An obvious candidate is the custom code generator for creating adapter skeletons when integrating other domain-specific languages. In conjunction with this work, a graphical editor is imaginable that allows to link DSL and Pivot Model concepts more intuitively than via the annotation mechanism proposed in Section 6.2.2.

In contrast to these rather technical topics, I would also like to show up a more theoretical research agenda. For instance, it may be worthwhile to investigate the semantics of mappings from a target language to the Pivot Model concepts. When does it actually make sense to provide such a mapping and are there possibilities to deduce a default mapping from the abstract syntax and static semantics of the DSL in question? Approaches to ontology merging might show up interesting perspectives. Ultimately, the aim should be to further minimize the effort required for integrating a DSL with model management languages including — but not limited to — OCL. This would enable less and less technically adapt users to participate in the system modeling process. After all, bringing together domain experts and software engineers is one of the central objectives of model-driven software development.

Another promising research goal is to examine the problem of OCL queries on the System layer more closely. In this report, I have only suggested some fundamental abstractions required to make OCL work on the *Execution Level* of an arbitrary DSL. However, concrete case studies are required to reveal general principles for the design of applications employing OCL on the System level. This applies both to the problem of constraints ensuring correct runtime behavior and transformations over data using QVT.

Hopefully, the ideas and impulses presented in this section (and this report in general) will stimulate further research into domain-specific modeling languages and their integration with standards-based (meta-) modeling environments. Today, we are still far away from a widespread adoption of model-driven approaches for general systems development. However, as methodologies and tools gradually mature, the vision of a software development world reaching the productivity of today's automobile industry might become a reality. Then, error-prone and laborious coding will be largely replaced by precise models, declaratively specified semantics and automatically generated implementations.

A Specification of Pivot Model Operations

The wellformedness rules and the abstract syntax mapping defined in the OCL 2.0 Specification employ a number of additional operations on UML metaclasses [OMG06c, p. 55]. Here, I provide definitions of these operations adapted to the Pivot Model and error-corrected in case the original specification was flawed.

```
context Namespace

-- Returns a Type in this Namespace with the given name.
def: lookupType(name : String) : Type =
   self.ownedType->any(t | t.name = name)

-- Returns a Namespace in this Namespace with the given name.
def: lookupNamespace(name : String) : Namespace =
   self.nestedNamespace->any(ns | ns.name = name)
```

Listing A.1: *Operations on Namespace*

```
context Type

-- Gives true for a type that conforms to another.
def: conformsTo(other: Type): Boolean =
    (self=other) or (self.allParents()->includes(other))

-- Returns all of the direct and indirect ancestors of a type
def: allParents(): Set(Type) =
    self.superType->union(self.superType.allParents())

-- Results in the most specific common supertype of two types.
def: commonSuperType (other : Type) : Type =
   Type::allInstances()->select (cst |
      other.conformsTo (cst) and
      self.conformsTo (cst) and
      not Type::allInstances()->exists (clst |
         other.conformsTo (clst) and
         self.conformsTo (clst) and
         clst.conformsTo (cst) and
         clst <> cst
      )
   )->any (true)

-- Returns all properties of this Type and its supertypes.
def: allProperties() : Sequence(Property) =
   self.ownedProperty->union(self.superType.allProperties())

-- Returns all operations of this Type and its supertypes.
def: allOperations() : Sequence(Operation) =
   self.ownedOperation->union(self.superType.allOperations())

```

```
-- Returns a Property of this Type with the given name.
def: lookupProperty(name : String) : Property =
   self.allProperties()->any(p | p.name = name)

-- Returns an Operation of this Type with the given name and
-- the given parameter types.
def: lookupOperation(name:String, paramTypes:Sequence(Type)):Operation =
   self.allOperations()->any (op | op.name = name and
      op.hasMatchingSignature(paramTypes))
```

Listing A.2: *Operations on Type*

```
context Enumeration

-- Returns an EnumerationLiteral of this Enumeration with the given name.
def: lookupLiteral(name : String) : EnumerationLiteral =
   self.ownedLiteral->any(l | l.name = name)
```

Listing A.3: *Operations on Enumeration*

```
context Property

-- Returns true if the compared property has identical name and type.
def: cmpSlots(p : Property): Boolean =
   p.name = self.name and p.type = self.type
```

Listing A.4: *Operations on Property*

```
context Operation

-- Checks whether the Operation's signature matches with a sequence of
-- types. Note that in making the match only parameters with direction
-- kind 'in' or 'inout' are considered.
def: hasMatchingSignature(paramTypes: Sequence(Type)) : Boolean =
   let sigParamTypes: Sequence(Type) =
      self.inParameter.union(self.inoutParameter).type in
   (
      ( sigParamTypes->size() = paramTypes->size() ) and
      ( Set{1..paramTypes->size()}->forAll ( i |
         paramTypes->at(i).conformsTo(sigParamTypes->at(i)) )
      )
   )
```

Listing A.5: *Operations on Operation*

```
-- Results in a Property with the same name, type, etc. as the parameter.
-- Required to create tuple types from the output parameters of an Op.
context Parameter::asProperty(): Property
   pre: -- none
   post: result.name = self.name
   post: result.type = self.type
   post: result.isOrdered = self.isOrdered
   post: result.isMultiple = self.isMultiple
   post: result.isUnique = self.isUnique
   post: result.isStatic = false
```

Listing A.6: *Operations on Parameter*

B Essential OCL Metamodel

For reasons of completeness, this section also contains a few diagrams that I used for illustration when discussing the adaptations of the Essential OCL metamodel in Section 6.1.1.

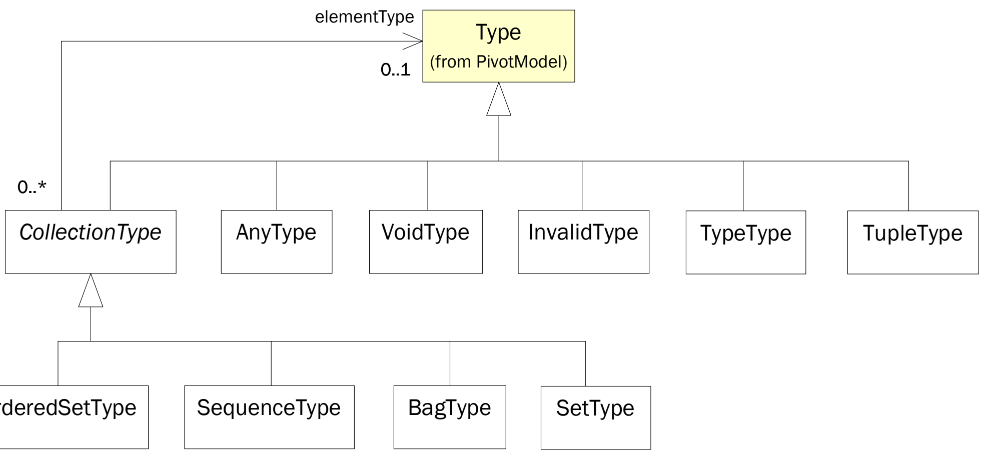


Figure B.1: *Types*

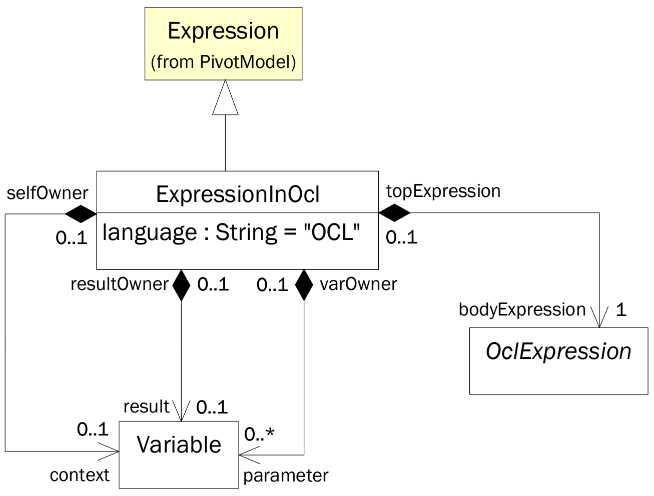


Figure B.2: *Top container expression*

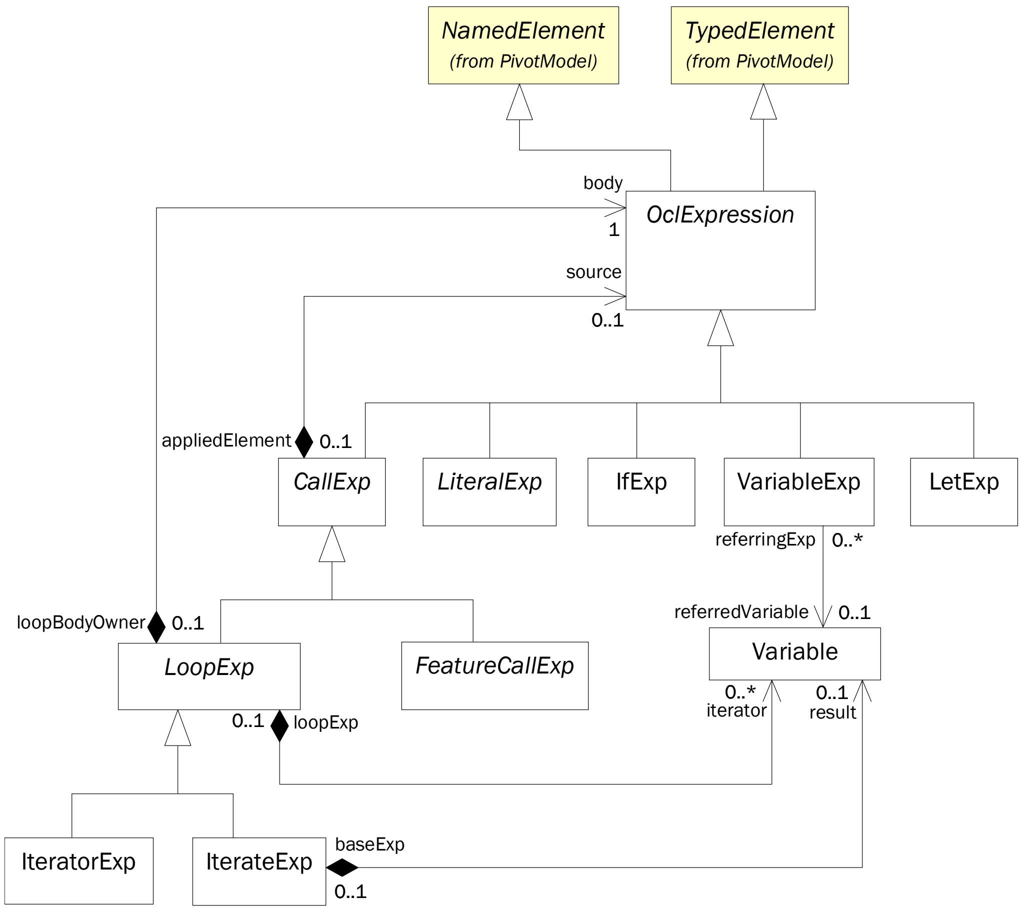


Figure B.3: *Main expression concept*

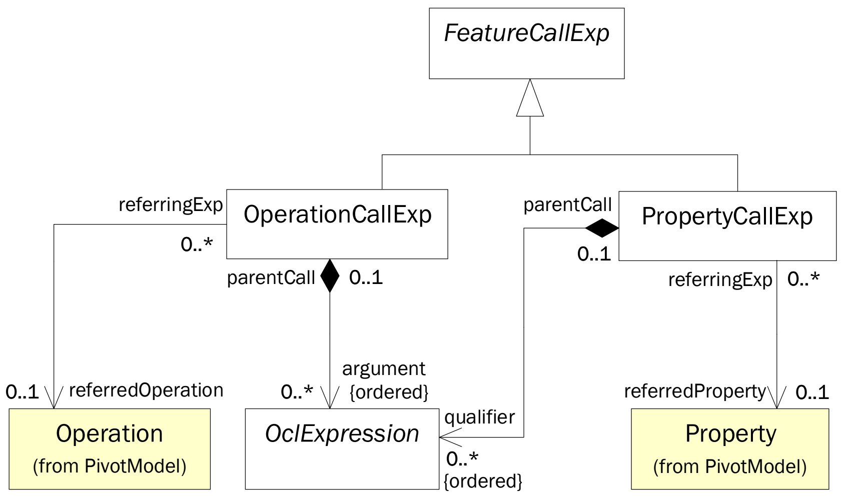


Figure B.4: *Feature Call expressions*

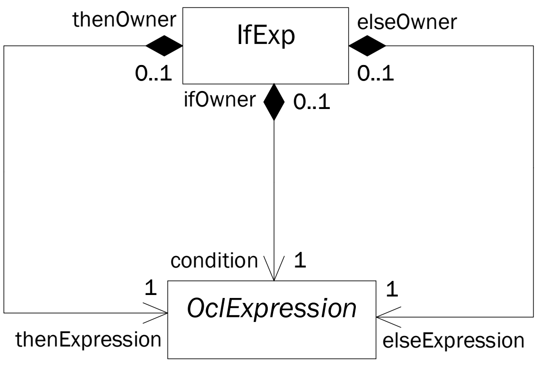


Figure B.5: *If expressions*

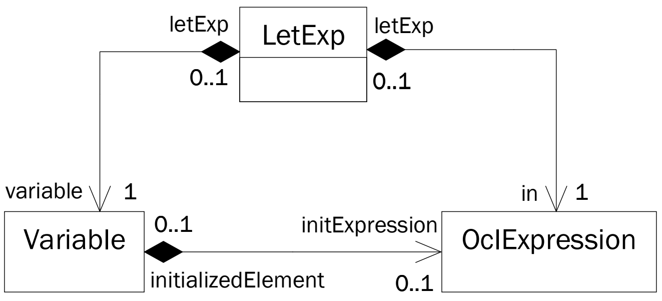


Figure B.6: *Let expressions*

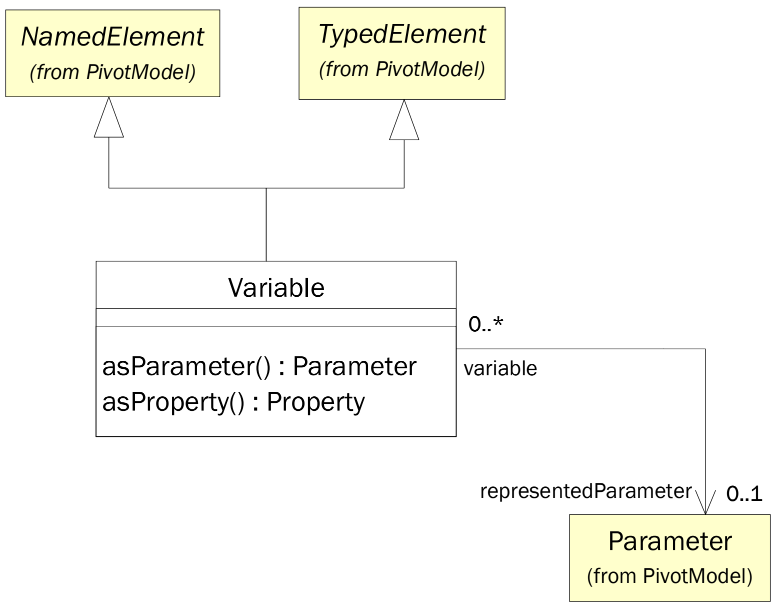


Figure B.7: *Variables*

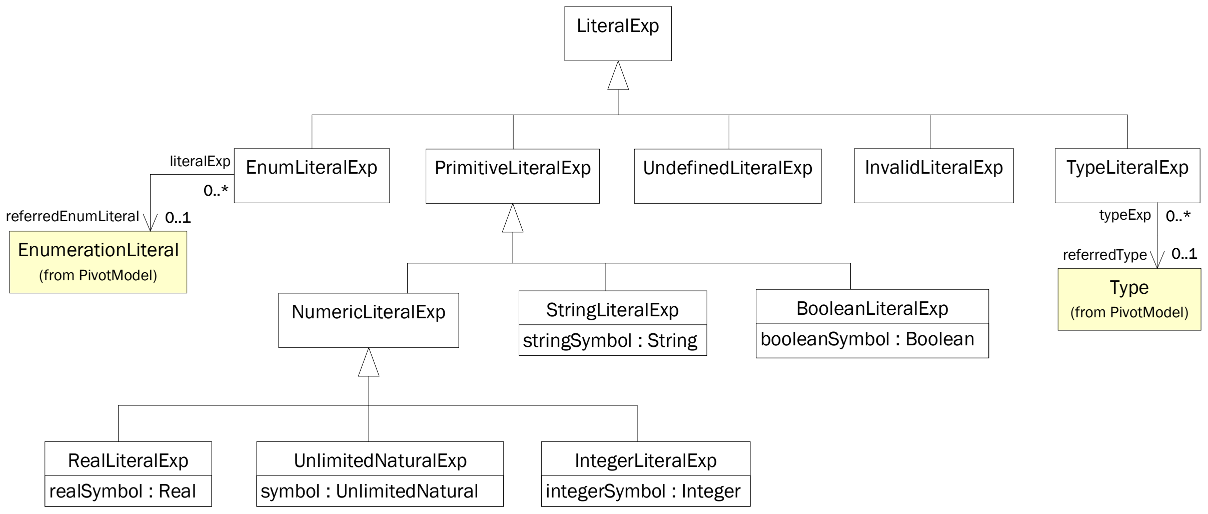


Figure B.8: *Literals*

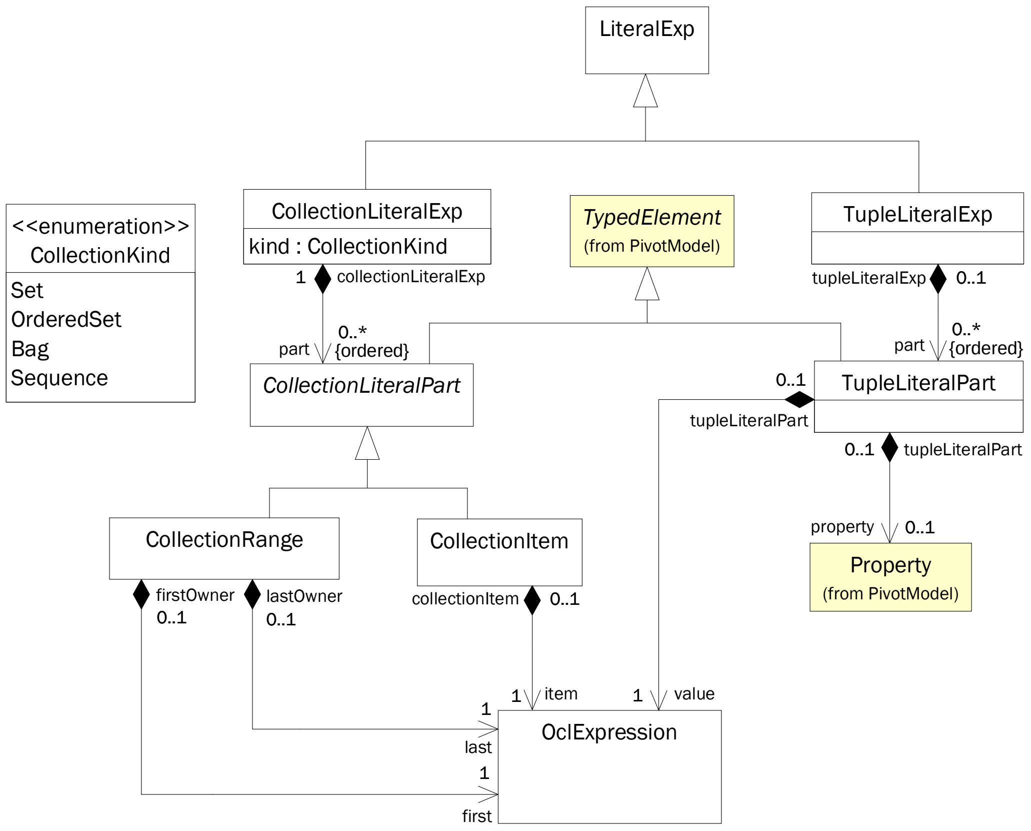


Figure B.9: *Collection and tuple literals*

C Specification of XOCL

This section fully specifies the abstract syntax of the XML-based OCL dialect called XOCL.

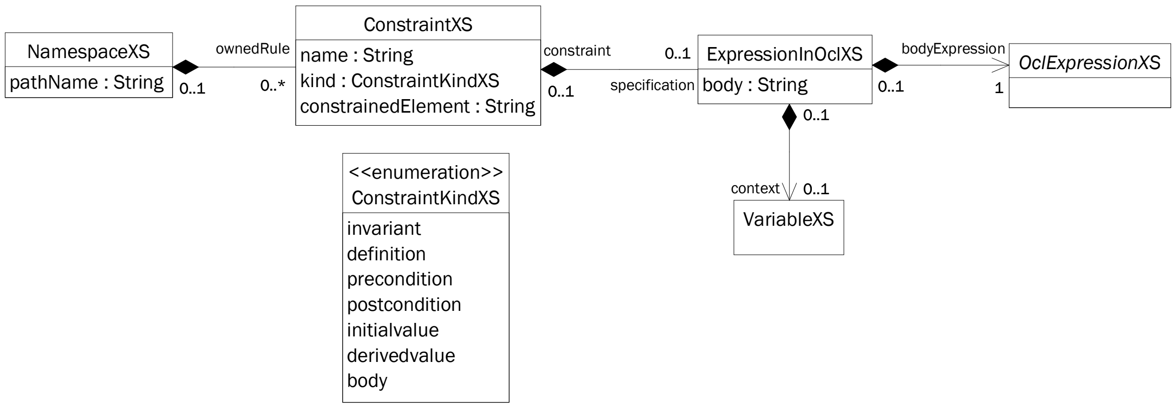


Figure C.1: *Constraints and top container expression*

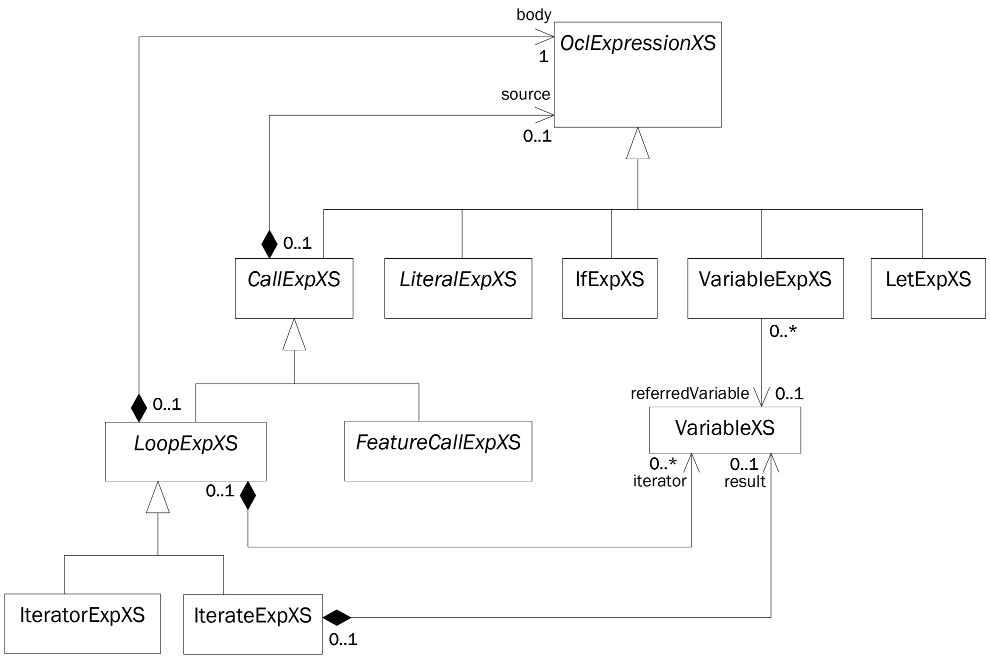


Figure C.2: *Main expression concept*

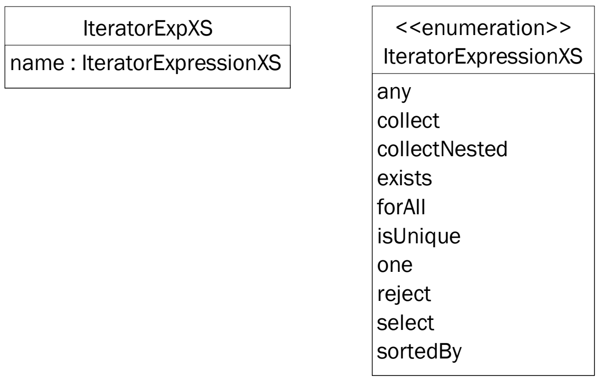


Figure C.3: *Iterator expressions*

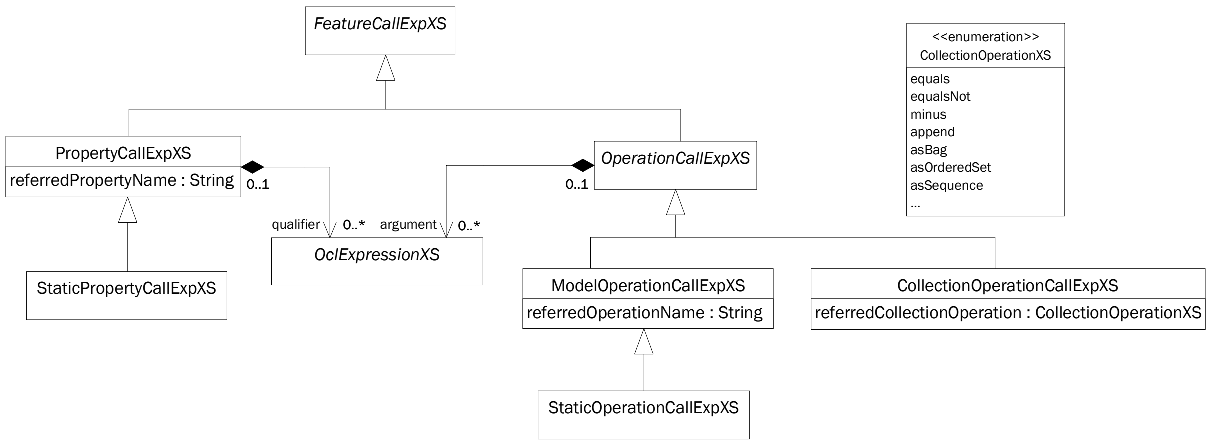


Figure C.4: *Feature Call expressions*

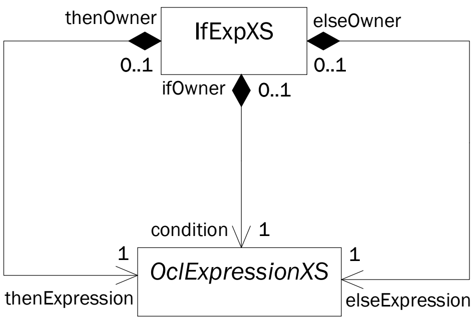


Figure C.5: *If expressions*

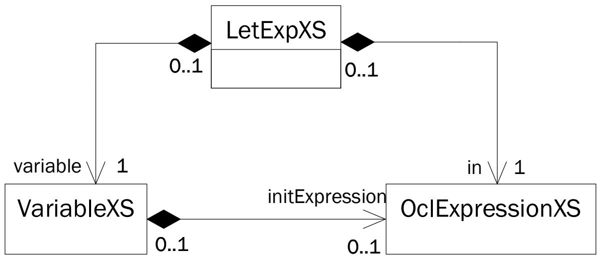


Figure C.6: *Let expressions*

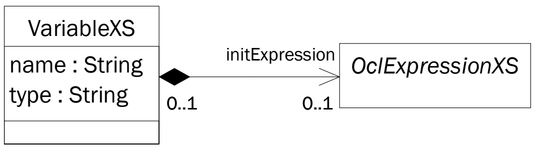


Figure C.7: *Variables*

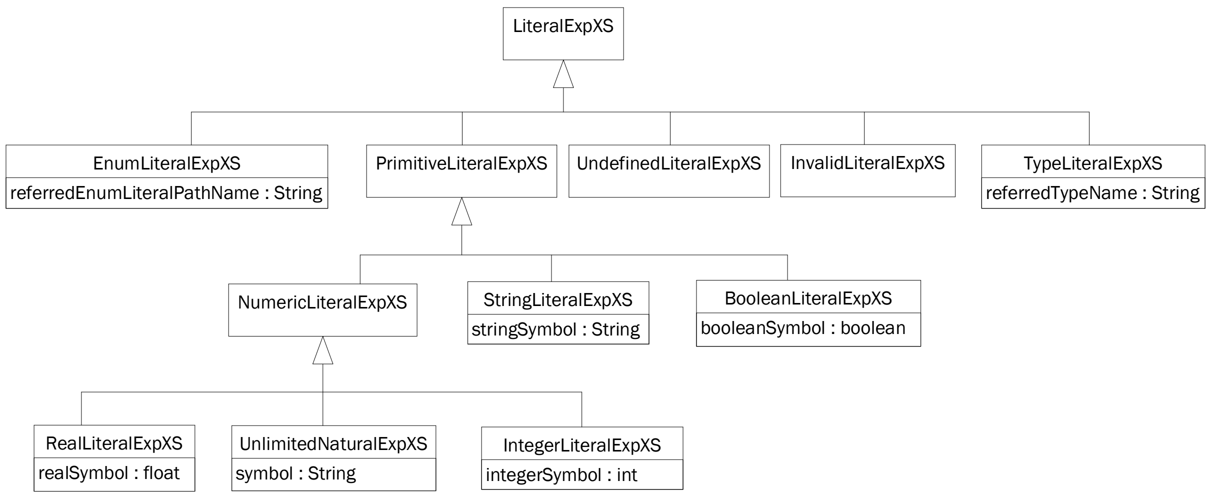


Figure C.8: *Literals*

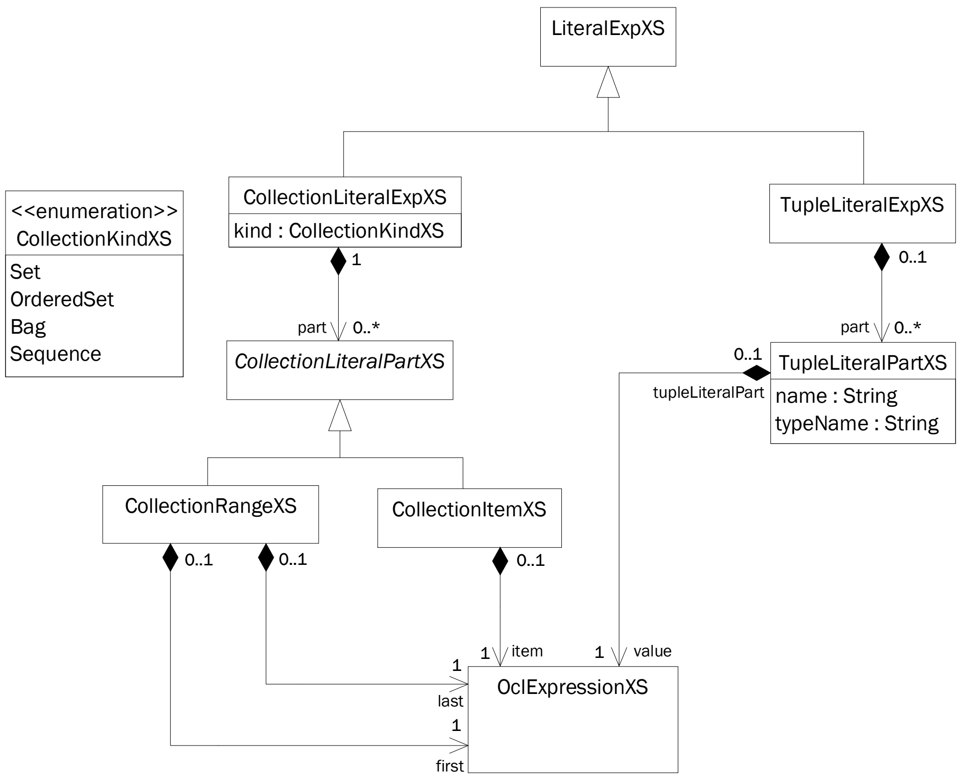


Figure C.9: *Collection and tuple literals*

List of Figures

List of Tables

Listings

List of Abbreviations

AMMA	ATLAS Model Management Architecture
AOSD	Aspect-Oriented Software Development
CWM	Common Warehouse Metamodel
DSL	Domain-specific (modeling) language
DSM	Domain-specific modeling
EMF	Eclipse Modeling Framework
FOP	Feature-Oriented Programming
GEF	Graphical Editing Framework
GME	Generic Modeling Environment
GMF	Eclipse Graphical Modeling Framework
GMT	Eclipse Generative Modeling Technologies
GPL	General-purpose language
GRS	Graph-Rewriting System
JET	Java Emitter Templates
JMI	Java Metadata Interfaces
JSP	Java Server Pages
KDM	Knowledge Discovery Metamodel
KM3	Kernel MetaMetaModel
KMF	Kent Modelling Framework
MDA	Model Driven Architecture
MDSD	Model-Driven Software Development
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Independent Model
PLE	Product Line Engineering
PML	Plugin Modeling Language

PSM	Platform Specific Model
QVT	Query / View / Transformation
SPEM	Software Process Engineering Metamodel
UML	Unified Modeling Language
XMI	XML Metadata Interchange

Bibliography

[AB01] D.H. Akehurst and B. Bordbar. On Querying UML Data Models with OCL. In *Proceedings of UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools: 4th International Conference*, volume 2185 of *LNCS*, pages 91–103, Toronto, Canada, October 2001. Springer.

[ABFJ05] Anas Abouzahra, Jean Bézivin, Marcos Didonet Del Fabro, and Frédéric Jouault. A Practical Approach to Bridging Domain Specific Languages with UML profiles. In *Proceedings of the Best Practices for Model Driven Software Development at OOPSLA'05*, San Diego, California, USA, October 2005.

[AHMM06] D.H. Akehurst, W.G.J. Howells, and K.D. McDonald-Maier. UML/OCL – Detaching the Standard Library. In *Proceedings OCLApps 2006: OCL for (Meta-) Models in Multiple Application Domains, MoDELS/UML 2006*, Technical Reports, pages 205–212, Genova, Italy, October 2006. Technische Universität Dresden, Fakultät Informatik.

[AK00] Colin Atkinson and Thomas Kühne. Meta-Level Independent Modeling. In *International Workshop Model Engineering (in Conjunction with ECOOP'2000)*, Cannes, France, June 2000.

[AK01] Colin Atkinson and Thomas Kühne. The Essence of Multilevel Metamodeling. In *Proceedings of UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools: 4th International Conference*, volume 2185 of *LNCS*, pages 19–33, Toronto, Canada, October 2001. Springer.

[AK02a] Colin Atkinson and Thomas Kühne. Rearchitecting the UML infrastructure. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 12(4):290–321, ACM Press, October 2002.

[AK02b] Colin Atkinson and Thomas Kühne. The role of meta-modeling in MDA. In Jean Bézivin and Robert France, editors, *International Workshop in Software Model Engineering (in conjunction with UML'02)*, Dresden, Germany, October 2002.

[AK03] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, IEEE Computer Society, 2003.

[ALP03] David Akehurst, Peter Linington, and Octavian Patrascoiu. OCL 2.0: Implementing the Standard. Computer Laboratory, University of Kent, November 2003.

[AMM] AMMA – ATLAS Model Management Architecture. Web page: `http://www.sciences.univ-nantes.fr/lina/atl`, INRIA, Université de Nantes. Last accessed: March 2007.

[AP04] David Akehurst and Octavian Patrascoiu. OCL 2.0 – Implementing the Standard for Multiple Metamodels. *Electronic Notes in Theoretical Computer Science*, (102):21–41, Elsevier, 2004.

[Aßm06] Uwe Aßmann. Design Patterns and Frameworks. Eclipse and Framework Extension Languages. Lecture Slides, 2006. Available at: `http://st.inf.tu-dresden.de/Lehre/WS05-06/dpf/slides/11-eclipse.pdf`.

[Bal00] Helmut Balzert. *Lehrbuch der Software-Technik — Band 1: Software-Entwicklung.* Lehrbücher der Informatik. Spektrum Akademischer Verlag, Heidelberg, 2nd edition, 2000.

[BBGN01] Don Batory, David Brant, Michael Gibson, and Michael Nolen. ExCIS: An Integration of Domain-Specific Languages and Feature-Oriented Programming. In *Workshop on New Visions for Software Design and Productivity: Research and Applications*, Nashville, Tennessee, December 2001. Vanderbilt University.

[BD07] Mariano Belaunde and Gregoire Dupe. SmartQVT - An open source model transformation tool implementing the MOF 2.0 QVT-Operational language. Web page: `http://smartqvt.elibel.tm.fr`, France Telecom R&D, 2007. Last accessed: April 2007.

[Béz05] Jean Bézivin. On the Unification Power of Models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proceedings of the 16th Conference on Automated Software Engineering*, pages 273–280, San Diego, USA, November 2001. IEEE Computer Society Press.

[BIR] Business Intelligence and Reporting Tools (BIRT) Project. Web page: `http://www.eclipse.org/birt`, Eclipse.org. Last accessed: April 2007.

[BJ06] Jean Bézivin and Frédéric Jouault. KM3: a DSL for Metamodel Specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 4037 of *LNCS*, pages 171–185, Bologna, Italy, 2006.

[BJKV06] Jean Bézivin, Frédéric Jouault, Ivan Kurtev, and Patrick Valduriez. Model-Based DSL Frameworks. In *Companion to the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006*, pages 602–616, Portland, Oregon, USA, October 2006. ACM Press.

[BL97] Jean Bézivin and Richard Lemesle. Ontology-Based Layered Semantics for Precise OA&D Modeling. In *Proceedings of the Workshops on Object-Oriented Technology. ECOOP'97 Workshop Reader*, volume 1357 of *LNCS*, pages 151–154, Jyväskylä, Finland, June 1997. Springer.

[Bra04] Gilad Bracha. Generics in the Java Programming Language. Sun Microsystems, July 2004. Available at: `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`.

[Bra06] Ronny Brandt. Java-Codegenerierung und Instrumentierung von Java-Programmen in der metamodellbasierten Architektur des Dresden OCL Toolkit. Großer Beleg, Technische Universität Dresden, Lehrstuhl für Softwaretechnologie, September 2006. In German.

[BSH99] Sjaak Brinkkemper, Motoshi Saeki, and Frank Harmsen. Meta-Modelling Based Assembly Techniques for Situational Method Engineering. In *10th International Conference on Advanced Information Systems Engineering (CAiSE'98), Pisa, Italy,* volume 24 of *Information Systems*, pages 209–228. Elsevier Science Ltd., June 1999.

[BSM+03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework.* Eclipse Series. Addison Wesley Longman, Amsterdam, 1st edition, August 2003.

[Bür05] Torsten Bürger. Contributions to Language Composition. Diplomarbeit, Technische Universität Dresden, Lehrstuhl für Softwaretechnologie, October 2005.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools and Applications.* Addison-Wesley Longman, Amsterdam, The Netherlands, 2000.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal, Special Issue on Model-Driven Software Development*, 45(3):621–645, IBM Corp., July 2006.

[CN01] Paul Clements and Linda M. Northrop. *Software Product Lines : Practices and Patterns.* Addison Wesley, 3rd edition, 2001.

[Coo06] Steve Cook. Domain-Specific Modeling. *The Architecture Journal*, 9:10–17, Microsoft Corp., 2006. Available at: `http://architecturejournal.net/2006/issue9/F2_Domain/`, Last accessed: January 2007.

[CR06] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-ins.* Eclipse Series. Addison-Wesley Longman, Amsterdam, 2nd edition, April 2006.

[DHK05] Birgit Demuth, Heinrich Hußmann, and Ansgar Konermann. Generation of an OCL 2.0 Parser. In Thomas Baar, editor, *Proceedings of the MoDELS'05 Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends*, number LGL-REPORT-2005-001 in Technical Reports, pages 38–52, Montego Bay, Jamaica, October 2005. Ecole Polytechnique Fédérale de Lausanne (EPFL).

[Dmi04] Sergey Dmitriev. Language Oriented Programming: The Next Programming Paradigm. White Paper, JetBrains, November 2004. Available at: `http://www.onboard.jetbrains.com/is1/articles/04/10/lop/`, Last accessed: January 2007.

[DR06] Hong-Hai Do and Erhard Rahm. Matching large schemas: Approaches and evaluation. *Information Systems*, Elsevier, October 2006. doi:10.1016/j.is.2006.09.002.

[Ecl] Eclipse Home. Web page: `http://www.eclipse.org`, Eclipse.org. Last accessed: April 2007.

[EMF] Eclipse Modeling Framework (EMF) Project. Web page: `http://www.eclipse.org/modeling/emf`, Eclipse.org. Last accessed: April 2007.

[Epsa] About Epsilon. Web page: `http://www.eclipse.org/gmt/epsilon/about.php`, Eclipse.org. Last accessed: March 2007.

[Epsb] Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon). Web page: `http://www-users.cs.york.ac.uk/~dkolovos/epsilon`, University of York. Last accessed: January 2007.

[ES06] Matthew Emerson and Janos Sztipanovits. Techniques for Metamodel Composition. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06)*, volume 37 of *Computer Science and Information System Reports, Technical Reports, University of Jyväskylä, Finland*, pages 123–139, Portland, Oregon, USA, October 2006.

[FBJ$^+$05] Marcos Didonet Del Fabro, Jean Bézivin, Frédéric Jouault, Erwan Breton, and Guillaume Gueltas. AMW: A Generic Model Weaver. In *Journées sur l'Ingénierie Dirigée par les Modèles (IDM05)*, Paris, France, June 2005.

[FBV06] Marcos Didonet Del Fabro, Jean Bézivin, and Patrick Valduriez. Weaving Models with the Eclipse AMW Plugin. In *Eclipse Modeling Symposium, Eclipse Summit Europe 2006*, Esslingen, Germany, October 2006.

[Fin99] Frank Finger. Java-Implementierung der OCL-Basisbibliothek. Großer Beleg, Technische Universität Dresden, Lehrstuhl für Softwaretechnologie, July 1999. In German.

[Fow04] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern. White Paper, ThoughtWorks, January 2004. Available at: `http://www.martinfowler.com/articles/injection.html`, Last accessed: April 2007.

[Fow05] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? White Paper, ThoughtWorks, June 2005. Available at: `http://www.martinfowler.com/articles/languageWorkbench.html`, Last accessed: January 2007.

[Fuj] Fujaba Tool Suite. Web page: `http://wwwcs.uni-paderborn.de/cs/fujaba`, Universität Paderborn. Last accessed: March 2007.

[FV07] Marcos Didonet Del Fabro and Patrick Valduriez. Semi-automatic Model Integration using Matching Transformations and Weaving Models. To appear in: The 22th Annual ACM Symposium on Applied Computing - Model Transformation Track (MT 2007), Seoul, Korea, 2007.

[Gam96] Erich Gamma. The Extension Objects Pattern. Submitted to PLoP'96, 1996.

[Gär06] Kai-Uwe Gärtner. Visualisierung des Abstrakten Syntaxmodells (ASM) von OCL-Ausdrücken. Großer Beleg, Technische Universität Dresden, Lehrstuhl für Softwaretechnologie, April 2006. In German.

[GEF] Graphical Editing Framework (GEF). Web page: `http://www.eclipse.org/gef`, Eclipse.org. Last accessed: April 2007.

[GFB05] Martin Gogolla, Jean-Marie Favre, and Fabian Büttner. On Squeezing M0, M1, M2, and M3 into a Single Object Diagram. In Thomas Baar, editor, *Proceedings of the MoDELS'05 Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends*, number LGL-REPORT-2005-001 in Technical Reports, pages 1–14, Montego Bay, Jamaica, October 2005. Ecole Polytechnique Fédérale de Lausanne (EPFL).

[GH05] Ralf Gitzel and Tobias Hildenbrand. A Taxonomy of Metamodel Hierarchies. Available at: `http://bibserv7.bib.uni-mannheim.de/madoc/volltexte/2005/993/`, April 2005.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison-Wesley Longman, Amsterdam, The Netherlands, 1995.

[GMF] Graphical Modeling Framework (GMF). Web page: `http://www.eclipse.org/gmf`, Eclipse.org. Last accessed: March 2007.

[GMT] Generative Modeling Technologies (GMT). Web page: `http://www.eclipse.org/gmt`, Eclipse.org. Last accessed: March 2007.

[GNR04] Emanuel S. Grant, Krish Narayanan, and Hassan Reza. Rigorously Defined Domain Modeling Languages. In Juha-Pekka Tolvanen, Jonathan Sprinkle, and Matti Rossi, editors, *Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM'04)*, number TR-33 in Computer Science and Information System Reports, Technical Reports, Vancouver, Canada, October 2004. University of Jyväskylä.

[Gog01] Martin Gogolla. Using OCL for Defining Precise, Domain-Specific UML Stereotypes. In Aybuke Aurum and Ross Jeffery, editors, *Proceedings 6th Australian Workshop on Requirement Engineering (AWRE'2001)*, pages 51–60. Centre for Advanced Software Engineering Research (CAESER), University of New South Wales, Sydney, Australia, 2001.

[GPFLC04] Asunción Gómez-Pérez, Mariano Fernández-López, and Oscar Corcho. *Ontological Engineering, with examples from the areas of Knowledge Management, e-Commerce and the Semantic Web.* Springer, Berlin, 2004.

[GPvS02] Giancarlo Guizzardi, Luis Ferreira Pires, and Marten J. van Sinderen. On the role of Domain Ontologies in the design of Domain-Specific Visual Modeling Languages. In Juha-Pekka Tolvanen, Jeff Gray, and Matti Rossi, editors, *Proceedings of 2nd Workshop on Domain-Specific Visual Languages (in conjunction with OOPSLA 2002)*, Seattle, WA, USA, November 2002. Centre for Telematics and Information Technology, University of Twente.

[GR98] Martin Gogolla and Mark Richters. On Constraints and Queries in UML. In Martin Schader and Axel Korthaus, editors, *Proceedings UML'97 Workshop 'The Unified Modeling Language - Technical Aspects and Applications*, pages 109–121. Physica-Verlag, Heidelberg, 1998.

[Gro06] Richard C. Gronback. Eclipse Modeling Project and OMG(tm) Standards. In *Eclipse Modeling Symposium, Eclipse Summit Europe 2006*, Esslingen, Germany, October 2006. Borland Software Corporation.

[GS04] Jack Greenfield and Keith Short. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley & Sons, Indianapolis, Indiana, USA, 2004.

[GT06] Richard C. Gronback and Artem Tikhomirov. Developing a Domain-Specific Modeler with the Eclipse Graphical Modeling Framework (GMF). European Conference on Object-Oriented Programming (ECOOP), 20th edition, Nantes, France, Tutorial Slides, Borland Software, July 2006. Available at: `http://wiki.eclipse.org/index.php/GMF_Documentation`, Last accessed: January 2007.

[Hei05] Florian Heidenreich. SQL-Codegenerierung in der metamodellbasierten Architektur des Dresden OCL Toolkit. Großer beleg, Technische Universität Dresden, Lehrstuhl für Softwaretechnologie, May 2005. In German.

[Hei06] Florian Heidenreich. OCL-Codegenerierung für deklarative Sprachen. Diplomarbeit, Technische Universität Dresden, Lehrstuhl für Softwaretechnologie, April 2006. In German.

[HKKR05] Martin Hitz, Gerti Kappel, Elisabeth Kapsammer, and Werner Retschitzegger. *UML @ Work: Objektorientierte Modellierung mit UML 2.* dpunkt.verlag, Heidelberg, 3rd edition, 2005. In German.

[HL06] Florian Heidenreich and Henrik Lochmann. Using Graph-Rewriting for Model Weaving in the context of Aspect-Oriented Product Line Engineering. In *Proceedings of the First Workshop on Aspect-Oriented Product Line Engineering (AO-PLE'06) co-located with the International Conference on Generative Programming and Component Engineering (GPCE'06)*, Portland, Oregon, USA, October 2006.

[HZ04] Heinrich Hußmann and Steffen Zschaler. The Object Constraint Language for UML 2.0 – Overview and Assessment. *UPGRADE – The European Journal for the Informatics Professional*, V(2):25–28, April 2004.

[IBM] Rational Rose Modeler Resource Page. Web page: `http://www-306.ibm.com/software/awdtools/developer/rose/modeler`, IBM Software. Last accessed: April 2007.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet

Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer.

[KMF] Kent Modelling Framework (KMF). Web page: `http://www.cs.kent.ac.uk/projects/kmf`, University of Kent at Canterbury, Department of Computing. Last accessed: March 2007.

[Kon03] Ansgar Konermann. Entwurf und prototypische Implementation eines OCL2.0-Parser. Diplomarbeit, Technische Universität Dresden, Lehrstuhl für Softwaretechnologie, August 2003. In German.

[KPKP06] Dimitrios S. Kolovos, Richard F. Paige, Tim Kelly, and Fiona A.C. Polack. Requirements for Domain-Specific Languages. In *1st ECOOP Workshop on Domain-Specific Program Development (DSPD), in conjunction with ECOOP 2006*, Nantes, France, July 2006.

[KPP06a] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Eclipse Development Tools for Epsilon. In *Eclipse Modeling Symposium, Eclipse Summit Europe 2006*, Esslingen, Germany, October 2006.

[KPP06b] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Merging Models with the Epsilon Merging Language (EML). In *Proceedings of MoDELS 2006 – 9th International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 215–229, Genova, Italy, October 2006. Springer.

[KPP06c] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. The Epsilon Object Language (EOL). In *Proceedings European Conference in Model Driven Architecture (EC-MDA) 2006*, volume 4066 of *LNCS*, pages 128–142, Bilbao, Spain, July 2006. Springer.

[KPP06d] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Towards Using OCL for Instance-Level Queries in Domain Specific Languages. In *Proceedings OCLApps 2006: OCL for (Meta-) Models in Multiple Application Domains, MoDELS/UML 2006*, Technical Reports, pages 26–37, Genova, Italy, October 2006. Technische Universität Dresden, Fakultät Informatik.

[Küh06] Thomas Kühne. Matters of (Meta-) Modeling. *Software and Systems Modeling*, 5(4):369–385, Springer, December 2006.

[KvdB04] Ivan Kurtev and Klaas van den Berg. Unifying Approach for Model Transformations in the MOF Metamodeling Architecture. In *Proceedings of the 1st European MDA Workshop, MDA-IA*. University of Twente, the Nederlands, March 2004.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained — The Model Driven Architecture: Practice and Promise.* Object Technology Series. Addison-Wesley Longman, Amsterdam, The Netherlands, 2003.

[LMB$^+$01] 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 – IEEE International Workshop on Intelligent Signal Processing*, Budapest, Hungary, May 2001. Vanderbilt University, Institute for Software Integrated Systems, IEEE Press.

[LO04] Sten Loecher and Stefan Ocke. A Metamodel-Based OCL-Compiler for UML and MOF. *Electronic Notes in Theoretical Computer Science*, (102):43–61, Elsevier, 2004.

[MC07] Domain-Specific Modeling with MetaEdit+: 10 Times Faster Than UML. White Paper, MetaCase Consulting, 2007. Available at: `http://www.metacase.`

com/papers/Domain-specific_modeling_10X_faster_than_UML.pdf, Last accessed: January 2007.

[MDR] Metadata Repository (MDR). Web page: http://mdr.netbeans.org, Netbeans.org. Last accessed: March 2007.

[MDTa] Model Development Tools (MDT). Web page: http://www.eclipse.org/modeling/mdt, Eclipse.org. Last accessed: April 2007.

[MDTb] Model Development Tools (MDT) OCL component. Web page: http://www.eclipse.org/modeling/mdt/?project=ocl, Eclipse.org. Last accessed: March 2007.

[MGG$^+$06] Milan Milanovic, Dragan Gasevic, Adrian Giurca, Gerd Wagner, and Vladan Devedzic. On Interchanging Between OWL/SWRL and UML/OCL. In *Proceedings OCLApps 2006: OCL for (Meta-) Models in Multiple Application Domains, MoDELS/UML 2006*, Technical Reports, pages 81–95, Genova, Italy, October 2006. Technische Universität Dresden, Fakultät Informatik.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

[MODa] MODELPLEX. Web page: http://www.modelplex-ist.org. Last accessed: March 2007.

[MODb] MODELWARE (IST Project 511731). Web page: http://www.modelware-ist.org. Last accessed: January 2007.

[MP07] Ed Merks and Marcelo Paternostro. Modeling Generics With Ecore. In *EclipseCon 2007, Santa Clara, California, USA*. IBM Corp., March 2007. Tutorial Slides. Available at: http://www.eclipsecon.org/2007/index.php?page=sub/&id=3845.

[MSUW04] Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled: Principles of Model-Driven Architecture.* Object Technology Series. Addision-Wesley Longman, Amsterdam, The Netherlands, 2004.

[MW94] Thomas J. McCabe and Arthur H. Watson. Software Complexity. *Journal of Defense Software Engineering*, 7(12):5–9, December 1994.

[MZ98] Tova Milo and Sagit Zohar. Using Schema Matching to Simplify Heterogeneous Data Translation. In *Proceedings of the 24th VLDB Conference*, pages 122–133, New York, USA, August 1998.

[NR68] Peter Naur and Brian Randell, editors. *Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968*, January 1968. Available at: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

[Ock03] Stefan Ocke. Entwurf und Implementation eines metamodellbasierten OCL-Compilers. Diplomarbeit, Technische Universität Dresden, Lehrstuhl für Softwaretechnologie, June 2003. In German.

[OMGa] Object Management Group (OMG). Web page: http://www.omg.org. Last accessed: May 2007.

[OMGb] UML 1.5 Models in MDL Format - Action Semantics FTF outcome. Download: http://www.omg.org/docs/ptc/02-09-04.zip, Object Management Group (OMG). Last accessed: January 2007.

[OMGc] UML 2.0 Superstructure FTF Rose model containing the UML 2 metamodel. Download: http://www.omg.org/docs/ptc/04-10-05.zip, Object Management Group (OMG). Last accessed: January 2007.

[OMGd] UML Resource Page. Web Page: http://www.uml.org, Object Management Group (OMG). Last accessed: January 2007.

[OMG02] Object Management Group – OMG. *Meta Object Facility (MOF) Specification, Version 1.4*, April 2002. OMG document number formal/02-04-03, Available at: http://www.omg.org/docs/formal/02-04-03.pdf.

[OMG03a] Object Management Group – OMG. *Common Warehouse Metamodel (CWM) Specification, Version 1.1, Volume 1*, March 2003. OMG document number formal/03-03-02, Available at: http://www.omg.org/docs/formal/03-03-02.pdf.

[OMG03b] Object Management Group – OMG. MDA Guide Version 1.0.1. June 2003. OMG document number omg/2003-06-01, Available at: http://www.omg.org/docs/omg/03-06-01.pdf.

[OMG04] Object Management Group – OMG. *Unified Modeling Language Specification, Version 1.4.2*, July 2004. OMG document number formal/04-07-02, Available at: http://www.omg.org/docs/formal/04-07-02.pdf.

[OMG05a] Object Management Group – OMG. *MOF 2.0/XMI Mapping Specification, Version 2.1*, September 2005. OMG document number formal/05-09-01, Available at: http://www.omg.org/docs/formal/05-09-01.pdf.

[OMG05b] Object Management Group – OMG. *Software Process Engineering Metamodel Specification, Version 1.1*, January 2005. OMG document number formal/05-01-06, Available at: http://www.omg.org/docs/formal/05-01-06.pdf.

[OMG05c] Object Management Group – OMG. *Unified Modeling Language: Infrastructure Specification, Version 2.0*, March 2005. OMG document number formal/05-07-05, Available at: http://www.omg.org/docs/formal/05-07-05.pdf.

[OMG05d] Object Management Group – OMG. *Unified Modeling Language: Superstructure Specification, Version 2.0*, August 2005. OMG document number formal/06-01-01, Available at: http://www.omg.org/docs/formal/06-01-01.pdf.

[OMG06a] Object Management Group – OMG. *Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM)*, June 2006. OMG document number ptc/06-06-07, Available at: http://www.omg.org/docs/ptc/06-06-07.pdf.

[OMG06b] Object Management Group – OMG. *Meta Object Facility (MOF) Core Specification, Version 2.0*, January 2006. OMG document number formal/05-07-04, Available at: http://www.omg.org/docs/formal/05-07-04.pdf.

[OMG06c] Object Management Group – OMG. *Object Constraint Language, Version 2.0*, May 2006. OMG document number formal/06-05-01, Available at: http://www.omg.org/docs/formal/06-05-01.pdf.

[PB03] Rachel A. Pottinger and Philip A. Bernstein. Merging Models Based on Given Correspondences. In *Proceedings of the 29th VLDB Conference*, pages 862–873, Berlin, Germany, September 2003.

[Pöt06] Julia Pötschke. Spezifikation und Implementierung einer Transformationsvorschrift für ein MOF-basiertes Meta-Modell auf eine Java-basierte Zielplattform. Diplomarbeit, Technische Universität Dresden, Lehrstuhl Rechnernetze, October 2006. In German.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual, Second Edition.* Object Technology Series. Addison-Wesley Longman, Amsterdam, The Netherlands, 2004.

[Sei03] Ed Seidewitz. What Models Mean. *IEEE Software*, 20(5):26–32, IEEE Computer Society, 2003.

[SOE02] Kirk Schloegel, David Oglesby, and Eric Engstrom. Towards Next Generation Metamodeling Tools. In Juha-Pekka Tolvanen, Jeff Gray, and Matti Rossi, editors, *Proceedings of 2nd Workshop on Domain-Specific Visual Languages (in conjunction with OOPSLA 2002)*, Seattle, WA, USA, November 2002. Aerospace Electronic Systems Research Lab, Honeywell International.

[Stölzel05] Mirko Stölzel. Entwurf und Implementierung der Integration des Dresden OCL Toolkit in Fujaba. Großer Beleg, Technische Universität Dresden, Lehrstuhl für Softwaretechnologie, November 2005. In German.

[Sun02] Sun Microsystems. *Java Metadata Interface (JMI) Specification*, June 2002. JSR 040, Java Community Process, Available at: `http://www.jcp.org/aboutJava/communityprocess/final/jsr040/index.html`.

[SV06] Thomas Stahl and Markus Völter. *Model-Driven Software Development: Technology, Engineering, Management.* Wiley & Sons, 1st edition, 2006.

[SZG06] Mirko Stölzel, Steffen Zschaler, and Leif Geiger. Integrating OCL and Model Transformations in Fujaba. In *Proceedings OCLApps 2006: OCL for (Meta-) Models in Multiple Application Domains, MoDELS/UML 2006*, Technical Reports, pages 140–150, Genova, Italy, October 2006. Technische Universität Dresden, Fakultät Informatik.

[TPT] Eclipse Test & Performance Tools Platform (TPTP) Project. Web page: `http://www.eclipse.org/tptp`, Eclipse.org. Last accessed: April 2007.

[TUD] Dresden OCL Toolkit. Web page: `http://dresden-ocl.sourceforge.net`, Technische Universität Dresden, Department of Computer Science. Last accessed: March 2007.

[VKEH06] Markus Völter, Bernd Kolb, Sven Efftinge, and Arno Haase. From Front End To Code - MDSD in Practice. Eclipse Foundation, June 2006. Available at: `http://www.eclipse.org/articles/Article-FromFrontendToCode-MDSDInPractice/article.html`.

[Völ05] Markus Völter. Patterns for Handling Cross-Cutting Concerns in Model-Driven Software Development. In *EuroPLoP 2005*, December 2005. Available at: `http://www.voelter.de/data/pub/ModelsAndAspects.pdf`.

[W3C07a] World Wide Web Consortium – W3C. *XML Path Language (XPath) 2.0*, January 2007. W3C Recommendation, Available at: `http://www.w3.org/TR/2007/REC-xpath20-20070123/`.

[W3C07b] World Wide Web Consortium – W3C. *XSL Transformations (XSLT) Version 2.0*, January 2007. W3C Recommendation, Available at: `http://www.w3.org/TR/2007/REC-xslt20-20070123/`.

[Wen06a] Christian Wende. Entwicklung eines konfigurierbaren Datenbankschemagenerators für das Dresden OCL2 Toolkit. Großer beleg, Technische Universität Dresden, Lehrstuhl für Softwaretechnologie, May 2006. In German.

[Wen06b] Christian Wende. Konzeption einer QVT Engine im Rahmen des Dresden OCL Toolkit. Diplomarbeit, Technische Universität Dresden, Lehrstuhl für Softwaretechnologie, December 2006. In German.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language Second Edition: Getting Your Models Ready for MDA.* Object Technology Series. Addison-Wesley Longman, Amsterdam, The Netherlands, 2nd edition, 2003.

[WK06] Jos Warmer and Anneke Kleppe. Building a Flexible Software Factory Using Partial Domain Specific Models. In *Proceedings of the 6th OOPSLA Workshop on*

Domain-Specific Modeling (DSM'06), volume 37 of *Computer Science and Information System Reports, Technical Reports, University of Jyväskylä, Finland*, pages 15–22, Portland, Oregon, USA, October 2006.

[WTP] Web Tools Platform (WTP) Project. Web page: `http://www.eclipse.org/webtools`, Eclipse.org. Last accessed: April 2007.

[WWW] Kent Object Constraint Language Library. Web page: `http://www.cs.kent.ac.uk/projects/ocl`, University of Kent at Canterbury, Department of Computing. Last accessed: March 2007.

[ZDD06] Alanna Zito, Zinovy Diskin, and Juergen Dingel. Package Merge in UML 2: Practice vs. Theory? In *Proceedings of MoDELS 2006 – 9th International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 185–199, Genova, Italy, October 2006. Springer.

[ZL01] Zheying Zhang and Kalle Lyytinen. A Framework for Component Reuse in a Metamodelling-Based Software Development. *Requirements Engineering*, 6(2):116–131, June 2001.