HybridMDSD: Multi-Domain Engineering with Model-Driven Software Development using Ontological Foundations

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Medien-Inf. Henrik Lochmann
geboren am 4. Januar 1981 in Berlin

Gutachter:
Prof. Dr. rer. nat. habil. Uwe Aßmann, Technische Universität Dresden
Associate Professor Kasper Østerbye, IT University of Copenhagen

Tag der Verteidigung: 21.12.2009

Dresden, im Februar 2010

TECHNISCHE
UNIVERSITÄT
DRESDEN

Confirmation

I confirm that I independently prepared this thesis with the title

HybridMDSD: Multi-Domain Engineering with Model-Driven Software Development using Ontological Foundations

and that I used only the references and auxiliary means indicated in the thesis.

Dresden, February 25, 2010

Dipl.-Medien-Inf. Henrik Lochmann

Acknowledgments

First and foremost a would like to thank my supervisor professor Uwe Aßmann for his motivation, constructive criticism, and his belief in my work. Once he was convinced by the topic, Uwe was a great sparring partner and excellent quality gate. Also, I would like to thank my second reviewer professor Kasper Østerbye, who proved to be very flexible in a tight schedule towards the submission deadline.

I thank my employer the SAP AG, which even enabled me to write a PhD thesis in context its PhD programm. As research associate in the german research project feasiPLe and participant in company-internal projects, I got a valuable insight into professional practice and challenges of large scale software engineering. I owe a great debt to my colleagues at SAP, especially Falk Hartmann, Patrick Wustmann, Steffen Göbel, and Kay Kadner, who never let me down when a topic needed to be discussed.

A very important source of criticism, constructive comments, and fruitful discussions were the irregular and regular meetings with the colleagues from the chair of software technology at TU Dresden. Their professional competence in diverse areas of technical and non-technical expertise helped a lot to create visions and reveal immature ideas. I thank all of you for your support.

Vital parts of this thesis were contributed by the diploma thesis students Matthias Bräuer, Manuel Zabelt, and Christian Hermann, who worked under my supervision. I had a lot of fun in our joint work and publications. I especially thank Matthias for his exemplary accuracy, unrestricted commitment, and brilliant mind, which brought forward the project significantly.

Thanks Florian Heidenreich, Björn Larsson, and Karin Fetzer for the invaluable comments on the written text. Your advices revealed logical inconsistencies, complicated verbalization, and inappropriate respresentations, and improved the quality of this thesis significantly.

Last but not least, special thanks go to my brother and my parents, who did not get tired in supporting me and always formed an indispensable backbone for my enthusiasm and endurance.

Henrik Lochmann
Dresden, February 2010

Abstract

Software development is a complex task. Executable applications comprise a mutlitude of diverse components that are developed with various frameworks, libraries, or communication platforms. The technical complexity in development retains resources, hampers efficient problem solving, and thus increases the overall cost of software production. Another significant challenge in market-driven software engineering is the variety of customer needs. It necessitates a maximum of flexibility in software implementations to facilitate the deployment of different products that are based on one single core.

To reduce technical complexity, the paradigm of *Model-Driven Software Development (MDSD)* facilitates the abstract specification of software based on modeling languages. Corresponding models are used to generate actual programming code without the need for creating manually written, error-prone assets. Modeling languages that are tailored towards a particular domain are called domain-specific languages (DSLs). *Domain-specific modeling (DSM)* approximates technical solutions with intentional problems and fosters the unfolding of specialized expertise. To cope with feature diversity in applications, the *Software Product Line Engineering (SPLE)* community provides means for the management of variability in software products, such as feature models and appropriate tools for mapping features to implementation assets.

Model-driven development, domain-specific modeling, and the dedicated management of variability in SPLE are vital for the success of software enterprises. Yet, these paradigms exist in isolation and need to be integrated in order to exhaust the advantages of every single approach. In this thesis, we propose a way to do so.

We introduce the paradigm of *Multi-Domain Engineering (MDE)* which means model-driven development with multiple domain-specific languages in variability-intensive scenarios. MDE strongly emphasize the advantages of MDSD with multiple DSLs as a neccessity for efficiency in software development and treats the paradigm of SPLE as indispensable means to achieve a maximum degree of reuse and flexibility. We present HybridMDSD as our solution approach to implement the MDE paradigm.

The core idea of HybidMDSD is to capture the semantics of particular DSLs based on properly defined semantics for software models contained in a central upper ontology. Then, the resulting semantic foundation can be used to establish references between arbitrary domain-specific models (DSMs) and sophisticated instance level reasoning ensures integrity and allows to handle partiucular change adaptation scenarios. Moreover, we present an approach to automatically generate composition code that integrates generated assets from separate DSLs. All necessary development tasks are arranged in a comprehensive development process. Finally, we validate the introduced approach with a profound prototypical implementation and an industrial-scale case study.

Contents

Chapter 1

Introduction

1.1 Motivation

In the recent decades, computer programs have become more and more complex. In the early beginnings of software industries, applications were written by specifying basic machine-level instructions that advised the computer to shift bits and bytes between hardware registers. During this time, transistors were as big as apples and bugs in computer programs were real insects that jammed the control flow by hiding in computer relays. Nowadays, a conventional home PC owns one or even several processors that hold many billions of transistors.

The steadily growing computer power led to software that equally became more and more powerful. Of course, applications consistently grew with every new feature and the development of software became more and more difficult to maintain. To manage the increased complexity in software development, new languages and tools were designed that alleviated program specification. First, assembly languages helped to simplify register operations. Later, programming on hardware level was replaced by new features like conditional expressions or procedures in languages, such as Fortran, Algol, or C. The introduction of *object-oriented programming (OOP)* changed software development in its foundations. Object-oriented *general purpose programming languages (GPLs)*, like Java or C++, provided new exciting possibilities with features like inheritance, encapsulation, and polymorphism.

The process of abstraction constantly evolved and was combined with the process of tailoring to particular domains of expertise. The employment of *domain-specific languages (DLSs)*, or 4th generation languages (4GL), in software development demarcates the actual end of the simplification evolution. Such languages allow the specification of executable programs in domain-specific terms and representations. Users do not need to be programming experts anymore but can concentrate on their profession in contributing solutions for particular concerns in development. Depending on the level of abstraction, a DSL might even allow to specify complete programs, without the need for any technical support. Currently, we can witness a trend towards the specification of arbitrary domain-specific languages and corresponding tooling. During the last few years, several frameworks emerged that allow to easily design languages from their basic structural and semantic foundations up to textual or graphical editors that facilitate convenient usage.

Just like the growth of technological possibilities, the software market evolved to a dynamic and highly competitive ecosystem. Especially the simplification process in software development and the voluntary formation of intellectual power in open source communities enables even individuals to develop and offer exciting applications with surprisingly small efforts in time and resources. Hence, the development of solutions for very specific user-centric problems is much less problematic than it was ever before. To allow also large-scale enterprises that develop huge monolithic applications, which commonly can not be easily shifted into the direction of specific concerns, to flexibly focus on customer needs, management of variability in software became a topic. According research and development efforts led to a software development paradigm called *software product line engineering (SPLE)*. In SPLE *variability* is a first class concern and challenges like the integration of software artifacts depending on presence or absence of product features are considered.

Both the trend of abstraction to gain domain specificity and the dedicated handling of variability are advantageous for software development. However, their integrated use is still problematic. Common software products are usually too complex to be specified with only one single DSL. Instead, several languages are necessary. To this end, the consistent integration of all employed DSLs poses major challenges. Without proper qualification of dependencies and interactions between languages, it is impossible to ensure integrity in a development landscape where artifacts of different languages are used in combination. Additionally, the information about the interplay of artifacts must be encoded implicitly in language interpreters or transformations that translate the artifacts into GPL programs.

Besides the problem of ensuring integrity in a multi-language scenario, additional problems arise when the variability dimension is introduced to software development. Here, a software system that previously passed as consistent and complete, is now subject to feature-driven modifications that disrupt integrity once more. Such disruptive modifications must be detected and resolved. In addition, every high-level product feature needs a specification that describes all its modifications to existing application code right down to the last detail. This is a time consuming and error-prone task that causes effort across various domain limits. Hence, the separation of concerns that is gained through the employment of DSLs cannot be guaranteed during development of product features anymore.

1.2 Scope

The overall goal of this thesis is to provide an approach to software development using multiple domain-specific languages that involve abstract model-driven technologies in variability-driven scenarios. We aim at a solution that provides support for fully integrated end-to-end scenarios, including language specification, modeling, and the derivation of consistent programming code that facilitates executable applications. In this respect, we focus on *generative scenarios* that involve code generators which produce runtime code instead of *interpretive scenarios* which comprise model interpreters.

Concerning the nature of treated domain-specific languages we limit our consideration to *technical* DSLs. Such languages involve technical concepts and relationships that may or may not be bound to dedicated realization platforms. The presented approach is not applicable for the integration of computational independent languages. With respect to the nature of the software that shall be developed, we focus our consideration on applications that involve components which are common to desktop and web applications, such as a user interface or data persistence. Although we want to emphasize that the presented work is explicitly *not* restricted to such kind of software, we are oblidged to confine our validation because of the complexity of the topic.

For the realization of the addressed DSL integration we set Semantic Web technologies. Although there are alternative ways to establish the necessary language connection foundation, this thesis evaluates the appropriateness of ontologies an logical deduction with Semantic Web reasoners and corresponding rule engines. Moreover, also code level integration is based on these foundations. Here, abstract code composition patterns are involved that are created on language level and applied for actual inter-model relations. In this respect, code composition patterns are created *manually* although an automatic derivation from semantic structures is imaginable.

1.3 Contributions

1.3.1 Ontology for Software Models

The connecting element between diverse DSLs in multi-DSL development environments is constituted with an ontology for software models. We performed a comprehensive analysis of related work in the area of top-level ontologies in context of information systems and derived a new upper ontology to represent the semantics commonly contained in abstract software specifications. To the best of our knowledge, an upper ontology that explicitly focuses on technical software models did not exist before.

1.3.2 Composition Automation

This thesis introduces a novel approach that facilitates the production of integration structures between arbitrary programming code. Our approach establishs a new abstraction layer above common code composition techniques. This way, traditional techniques that were developed for injection, modification, or parameterization on code level reach a new level of value for software production.

1.3.3 Development Process for Multi-DSL Development

Software development with multiple DSLs is performed in various different steps where diverse development roles are involved. In literature, several development processes exist but none of them explictly focusses on actual end-to-end multi-DSL scenarios in variability-centric scenarios. There are several examples for engineering processes with multiple languages which do not include code generation or even the challenging task of code composition. Other examples precisely define development steps in product line engineering while not treating model-driven technologies. This thesis provides a comprehensive development process including all together with strictly delimited responsibilities, asset types, and performed tasks.

1.3.4 Comprehensive Case Study

Finally, another important contribution of this work is the development of a comprehensive industrial-scale case study that validates the presented approach and is the basis for discussion. Here, we provide a scenario that comprises six different domain-specific languages with sophisticated code generators, language level connections, code composition patterns, integrity constraints, and adaptation rules.

1.4 Organization

In this section we explain how the present thesis is organized. To this end, we clarify the typographical conventions and present the thesis' content structure.

1.4.1 Formatting Conventions

The remainder of this document is formatted as follows:

- *accentuations* are emphasized with italic font
- `code snippets` are rendered in type writer font
- diagram elements are indicated with sans serif font

1.4.2 Outline

The content part of this thesis starts with Chapter 2 that provides valuable background information, which is necessary to understand the following chapters. To this end, we comprehensively introduce into the involved paradigms and technologies, namely model-driven software development, domain-specific modeling, software product line engineering, and Semantic Web technologies.

In Chapter 3 we analyze the thesis' problem domain and sketch an appropriate vision that shall alleviate identified challenges. The chapter closes with a documented enumeration of essential requirements that need to be fulfilled to realize the outlined vision.

Chapter 4 gives a rough overview of the solution approach we follow to realize our vision. Here, we answer to previously stated requirements and introduce the solution architecture. Moreover, we present an illustrative example that shall help to understand intention.

Related approaches are reviewed in Chapter 5 before the actual results of this PhD thesis are presented in Chapter 6. The presentation is ordered like the thesis contributions stated above. Thus, we firstly introduce the semantic foundation, followed by an explanation of the code composition framework, and concluded with a description of our development process.

In Chapter 7 we validate our work with the developed prototype and the performed case study. In addition, we illustrate the application of dedicated solution features and discuss our findings. Chapter 8 summarizes this work and gives an outlook to future challenges.

Chapter 2

Background

This chapter provides background knowledge for the remainder of this thesis. In detail, we will have a look at three promising software development paradigms that provide potential for the advance of software engineering, namely *Model-Driven Software Development (MDSD)*, *Domain-Specific Modeling (DSM)*, and *Software Product Line Engineering (SPLE)*. In Chapter 3, we will review these paradigms and motivate for their integration. In addition to the paradigm introduction, we will give an overview of *Semantic Web* technologies which will be of importance for the explanations of our solution approach presented in Chapters 4, 6 and 7.

2.1 Model-Driven Software Development

2.1.1 Overview

Raising the level of abstraction proved effective to augment efficiency and explanatory power in software development several times in computer history. This way, programming languages evolved from pure machine-readable instructions to commonly employed object-oriented *general-purpose programming languages (GPLs)*, such as C++ or Java. *Model-Driven Software Development (MDSD)* is a paradigm which follows this trend and generalizes the historically grown abstraction process. It promises to increase productivity, enhance software quality and improve manageability of large software projects [Bet04, SVEH07]. Abstract models provide a less-detailed view on programming code. To this end, models narrow the developer's mental conceptualization, allowing to specify *what* a system should do rather than *how* it should do it [AK03]. This facilitates intuitive and fast implementation of complex software systems.

In MDSD, models are promoted to first class development constructs, which are commonly furnished with generators that produce executable programming code. This way, a model is not only used for documentation but rather denotes an integral part of software system specifications. Through code generation, it becomes possible to make use of arbitrary implementation technologies and various design patterns, that are only specified once. Readily developed code generators can be reused steadily and the risk to produce erroneous programming code is massively reduced. Figure 2.1 illustrates the general concepts of the MDSD paradigm.

2.1.2 Metamodeling

Model-driven development handles the description of existing information using additional metadata. Metamodeling, in this respect, is the process of analysing a domain in order to find

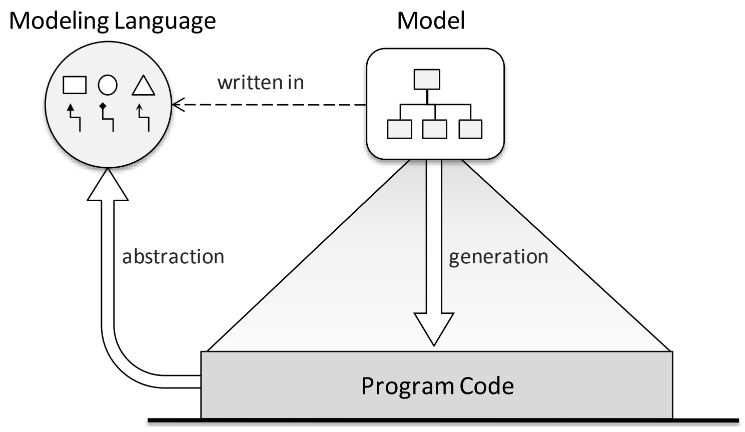


Figure 2.1: *Model-driven software development concepts*

concepts, relationships, rules and constraints for that domain, as well as the formalization of these findings in a metamodel. Therefore, metamodeling is an essential development step in model-driven development.

To understand the relation between actual information, metadata that describes this information and metamodels that prescribe the structure of models, we will introduce the important terms of this context within this section.

2.1.2.1 Models

In MDSD, models designate the actual metadata that describes information of interest. Because model-driven development considers the specification of software, the *modeled* information is usually about technical knowledge in the context of computer programs. This is in line with Kühne, who states that a "[...] model is an abstraction of a (real or language-based) system allowing predictions or inferences to be made" [Küh06, p. 1]. A more general definition on models is given by Völter and Stahl, who comprehend models as "[...] collections of data that adhere to a particular structure" [SV06, p. 59]. In principle, it is perfectly legal to describe arbitrary real world concepts and relationships with models, i.e., also domains that are not related to software development (viz. Section 2.1.2.3).

However, the key issue on models for MDSD is the fact that therein contained information "can be used to automate various aspects of the software development process" [GS04, p. 217]. Hence, modeled information must be processable by tools. To this end, models in MDSD must be *formal*, i.e., they are obliged to strictly adhere to the structure and well-formedness of the constraints described in a higher level definition that is given with a metamodel. At this point, models in MDSD clearly differ from metadata that describes a domain informally, such as whiteboard charts or scratch paper drawings. However, although the structure of models is fixed, their semantics is usually not formally specified [Ant07] (cf. Section 2.1.2.2).

2.1.2.2 Metamodels

Metamodels specify a set of modeling primitives and rules for valid combinations of these primitives in context of a particular domain. In this respect, metamodels "[...] define the abstract syntax of modeling languages" [GS04, p. 289] and their static semantics. In line with [AZW06] and [Gui07], we consider metamodels as prescriptive specifications of a domain whose main purpose is to specify a language for metadata (i.e. models) that shall assert concrete information about that domain. To this end, there is a clear emphasize on usability and intuitivity of metamodels. This indicates that metamodels do not claim to be *complete* w.r.t. a domain (cf. [Gui07]) and usually represent a particular extract of that domain. Therefore, they define at least several concepts and relationships that commit to the conceptual model of that domain, i.e., to its ontology (see 2.4). These concepts and relationships then belong to the *ontological commitment* of the metamodel (see Section 2.4.1.1).

Abstract Syntax and Static Semantics

As stated above, a metamodel is comprised of (1) abstract syntax and (2) static semantics. (1) specifies a set modeling primitives that may be used during modeling. In difference to the *concrete syntax* of a language, the abstract syntax specifies generally available language concepts, without defining their actual representation. (2) denotes well-formedness rules and constraints, which specify how modeling primitives are allowed to be composed. A prominent language to specify constraints that go beyond the expressiveness of an according meta-metamodel, is the *Object Constraint Language (OCL)* [OCL06, WK03]. OCL is an imperative constraint language to specify queries and invariants over model instances, which was released by the *Object Management Group (OMG)* [OMG].

Dynamic Semantics

The dynamic semantics of a modeling language cannot be specified with its metamodel. Although the metamodel prescribes how valid "sentences" of a modeling language must look like, the actual semantics of these sentences remains blurred. This is because the modeled information is still only metadata about certain system knowledge. What the actual meaning of this metadata constitutes is subject to interpreters and code generators, which convert the constructs of a modeling language into another language that has a well defined formal semantics. This kind of semantics is called *translational semantics* [GS04, p. 310].

2.1.2.3 MOF Metadata Architecture

During the last decade, there have been several efforts to develop standardizations in the area of MDSD. An important consortium, in this respect, is the OMG, which released a number of standards around model-driven development [CWM03, MDA03, QVT07, ODM06, OCL06], among others the *Meta Object Facility (MOF)* [MOF06a, MOF06b] specfication. One key contribution of this specification is the clarification of participating information layers in common model-driven scenarios. In its early version 1.4 the OMG described classical "four layer metadata architectures" [MOF06a, p. 9]. According to this, the metamodeling process is performed on four different layers of information, where each layer is used to describe the knowledge of its underlying sibling. This general metamodeling framework was used to derive the *MOF Metadata Architecture*, illustrated in Figure 2.2.

As depicted, the layers are numbered from one to three according to their distance from the actual information that is subject to specification. The M0 Layer of the stack which is omitted in the figure represents the data that is to be described. Instead of M0 Figure 2.2 shows the Code Layer that represents the runtime artifacts which constitute the information to handle. The M1 Layer denotes the modeling layer, i.e., the metadata for this information, which is prescriptively specified by the M2 Layer. The M2 Layer in turn is itself specified by the M3 Layer that contains the meta-metamodel. In the MOF specification, the terms "layer" and "level" are alternately used [MOF06a, p. 11]. In this thesis, we will from now on use the terms *level*, *level Mx*, or *meta level x* to identify the diverse MOF layers.

For better understanding, consider the two example columns in the Figure 2.2. The first

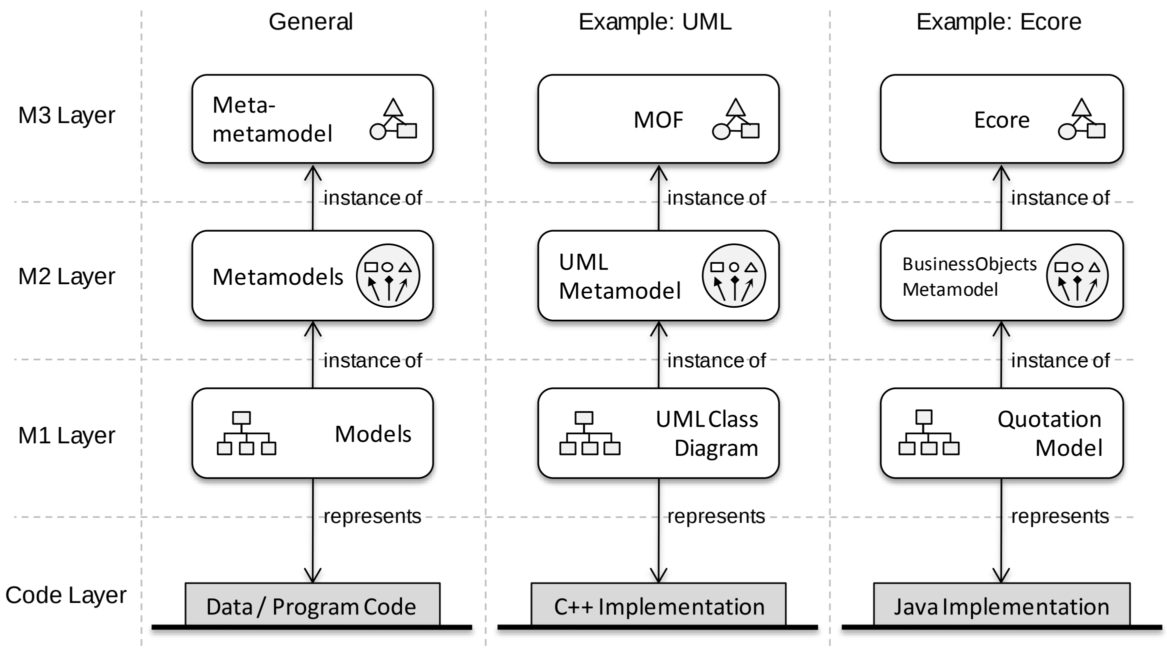


Figure 2.2: *MOF four layer metadata architecture*

example column shows a modeling stack for the *Unified Modeling Language (UML)* [UML07]. It illustrates a C++ implementation on code level which is specified with UML class diagrams on level M1. These diagrams are part of the UML metamodel that are specified on the basis of the MOF meta-metamodel [MOF06b]. The second example column shows an excerpt of our validation case study (see Section 7). Illustrated is a domain-specific language stack (cf. Section 2.2.2) for so-called BusinessObjects, a representation for data structures. The corresponding metamodel is an instance of the Ecore meta-metamodel [BSM+04] and has several instances, such as the depicted model for quotations in a modeling landscape for business applications.

Although the OMG revoked to strictly adhere to exactly four layers in a later version of MOF [MOF06b, pp. 8-9], the use of the proposed metamodeling architecture is widely accepted and often referred to in various publications [AK03, AZW06, Kur07].

2.1.2.4 Model Transformations

Model transformations are an important element of model-driven software development. They allow to convert models from one representation into another. One distinguishes between *model-to-model (M2M)* and *model-to-code (M2C)* transformations [CH03][SVEH07, p. 33]. M2M transformations convert a source model into a target model, where both do not necessarily adhere to the same metamodel. M2C transformations convert abstract models into textual representations of particular programming languages. In fact, M2C transformations can be considered as a special type of M2M transformations, simply lacking a metamodel for the target programming language [CH03]. However, for practical reasons, like execution environment specific details (e.g., precompiler statements), it makes sense to go without a dedicated target metamodel and preserve the freedom to generate arbitrary textual fragments. This is why M2C transformations are also called *M2T (model-to-text)* transformation.

Greenfield and Short generally differentiate between *vertical*, *horizontal* and *oblique transformations* [GS04, p. 456]. The terms vertical and horizontal refer to the level of abstraction, where vertical transformations convert between different levels and horizontal transformations remain at the same level. Oblique transformations, instead, encapsulate a sequence of transformations in one single transformation step. Compilers, for example, perform several horizontal transformations against the *Abstract Syntax Tree (AST)* of a programming language, before generating executable byte code. Vertical transformations, which are also referred to as *refinements*, are additionally subdivided into:

Specialization which produces more specialized specifications from less specialized specifications. For example, models of a language for business modeling, including data entities and behavioral services, might be transformed into models of a language for customer relationship modeling, which refines the base language with domain-specific constructs, like customer or quotation.

Elaboration which produces more detailed specifications from less detailed specifications. An example would be the transformation from an abstract user interface description, lacking dedicated style information, to a concrete user interface model, which provides information for particular visual details, such as parameters for widget extents, margins or widget coloring.

Realization which produces implementations from existing configurations, while these implementations must not necessarily be executable. Prominent examples are transformations from abstract platform-independent models to concrete platform-specific models (see Section 2.1.3.2).

Derivation which produces new configurations from existing configurations.

Decomposition which produces new configurations from individual objects of specifications.

A prominent example for a horizontal transformation is *refactoring*, where source and target metamodel remain the same. In case source and target metamodels are different from each other, a horizontal transformation implements *language migration* [MG06]. Table 2.1 summarizes different horizontal and vertical transformation types, where *endogenous* transformations remain in the same solution domain or representation format, while *exogenous* transformations do not.

	horizontal	**vertical**
endogenous	Refactoring	Formal refinement
exogenous	Language migration	Code generation

Table 2.1: *Orthogonal model transformation types (adapted from [MG06])*

Figure 2.3 illustrates the different transformation types along an abstraction layer stack. In the following we present more details for M2C and M2M transformations.

Model-to-code transformations

Model-to-code transformations denote what is commonly called *code generation*. Czarnecki et al. distinguish visitor-based approaches from template-based approaches [CH03]. The former provides “some visitor mechanism to traverse the internal representation of a model” and “write code to a text stream” [CH03, p. 9], while the latter uses predefined textual templates, which define slots that point to particular model entities and are substituted by an appropriate generator engine with corresponding content of input models. The visitor-based approach allows to imperatively implement transformations in object-oriented GPL frameworks. On the one hand, this allows to stay in environments that are familiar to developers. On the other hand, resulting transformations are very verbose, because each and every string of the desired output needs to be delivered programmatically. The template-based approach forces developers to choose a par-

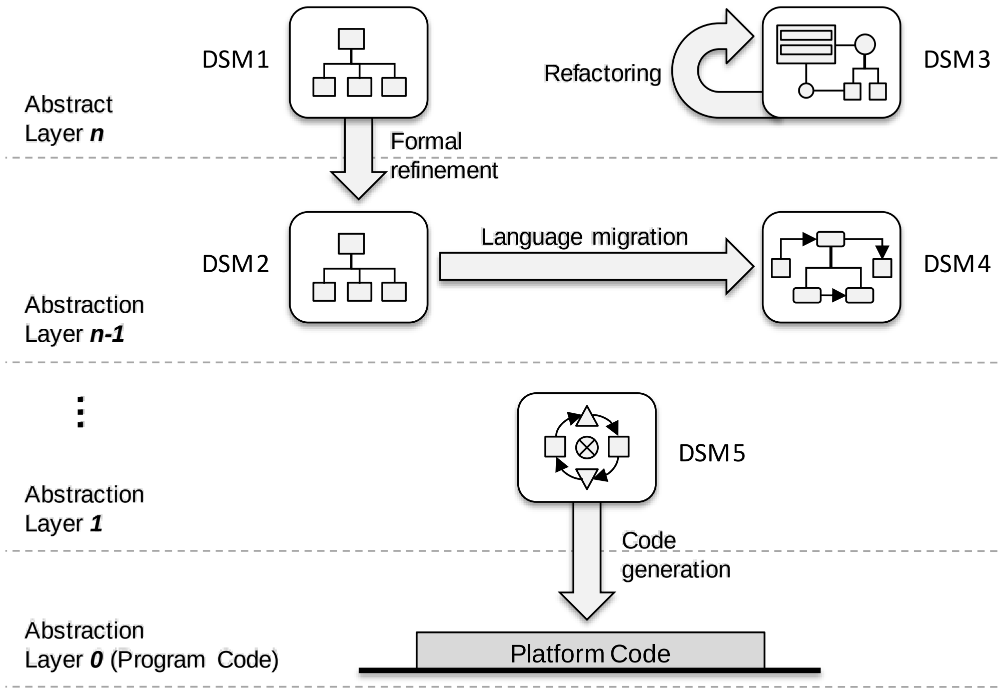


Figure 2.3: *Vertical and horizontal model transformations*

ticluar transformation engine. While this approach is much less verbose, developers are exposed to the flaws of these engines.

Today, many code generators exist, differing in application contexts, supported target programming languages and the degree of applicability [Dal09]. A code generation framework that enjoys lively support and ongoing development is *openArchitectureWare (OAW)* [ot09]. oAW is geared to facilitate code generation for Ecore-based metamodels (see Section 2.1.2.5) and is implemented as an Eclipse plugin [Ecl09]. An example for a code generation engine for MOF-based metamodels is MOFScript [Old06]. For a complete list of currently available code generators, the reader is referred to [Dal09].

Model-to-model transformations

Model-to-model transformations can be classified comparable to the previously described distinction between M2C transformations: the way *how* a transformation is specified can be used for differentiation. Thus, a top level criteria is whether a transformation specification is provided in *declarative* fashion or in an *imperative* language. Based on these two criteria, further classification can be performed according to the theoretical foundations employed for implementation. According to [CH03], the following M2M transformation types exist:

Relational Approaches belong to declarative transformation approaches and are based on mathematical relations. Transformations are specified by starting from source and target type and elaborating appropriate constructs using additional constraints. Relational approaches are side-effect-free and target elements are created implicitly.

Graph-Transformation-Based Approaches rely on the theoretical foundations of graph transformations that operate on typed, attributed, labeled graphs. The core idea is to define declarative rules, composed of *left-hand side (LHS)* and *right-hand side (RHS)* graph patterns. During execution, the corresponding engine tries to match the LHS in an input graph and exchanges matching structures with the RHS.

Structure-Driven Approaches are performed in two phases. First, the hierarchical structure of the target model is created. Second, appropriate references and attributes for the target are set. Such imperative approaches are commonly based on frameworks, i.e., the transformation developer must not struggle with details, such as scheduling of rule execution [CH03].

Hybrid Approaches summarize transformation approaches that combine aspects of all previous categories.

In recent years, several model-transformation languages evolved. A well-known is the *ATLAS Transformation Language (ATL)* [JK05], provided by the AMMA Platform (see next section), which is available as an Eclipse plugin. As a hybrid transformation language, it combines imperative and declarative techniques with OCL. An important OMG standardization for model-transformations is the *Query/View/Transformations (QVT)* specification [QVT07]. It defines a comprehensive model transformation framework, including an imperative part (QVT operations) and a declarative part (QVT relations). To find more examples for the summarized M2M transformation types, the reader is referred to [CH03].

2.1.2.5 Modeling Workbenches

Today, many tools and frameworks are available that provide convenient support for model-driven software development. Many of them emerged on top of the famous Eclipse *Integrated Development Environment (IDE)* [Ecl09], which provides rich extensibility mechanisms due to its sophisticated modularization architecture based on the OSGi service platform specification [OSG07]. In the following we present selected examples for different modeling workbenches.

The Eclipse Modeling Framework (EMF) is a very popular tool suite. Based on its meta-metamodel Ecore, an implementation nearly equivalent to the *Essential MOF (EMOF)* standard [Mer07, MOF06b], it provides comprehensive tooling to generate Java APIs for in-memory model management, model editors, and code generators. Additional projects and an active community around EMF ensures rich support when developing model-driven applications with this tool suite. The *Graphical Modeling Framework (GMF)* [Fou09e, KW06], for instance, facilitates the specification of visual editors for Ecore-based models or the oAW code generation framework enables for advanced code generation.

The AMMA Platform is provided as *ATLAS Model Management Architecture (AMMA)* [INR09] by the ATLAS group, a research team from the national institute for research in informatics and automation (INRIA) at the University of Nantes. The ATLAS group became previously known with its *ATLAS Transformation Language (ATL)*, a language for transformations between models based on the framework's meta-metamodel KM3 (Kernel MetaMeta-Model) [JB06]. More recently, several promising tools emerged around the AMMA architecture, such as the *ATLAS Model Weaver (AMW)* [FBJ$^+$05], *ATLAS MegaModel Management (AM3)* [ABBJ06] and *ATLAS Technical Projectors (ATP).*

The Generic Modeling Environment (GME) is the only here presented modeling workbench that is not Eclipse-based [LMB$^+$01, MBAL07]. The toolkit proves the tradition and experience of the *Institute for Software Integrated Systems (ISIS)* of the University of Vanderbilt in context of modeling technologies. GME allows for the creation of domain-specific modeling environments. To this end, the corresponding metamodeling language is based on the UML class diagram notation together with OCL constraints.

The Generic Eclipse Modeling System (GEMS) is also an Eclipse-based modeling workbench, which aims at bridging the gap between existing modeling experience achieved in previous projects, such as the GME project, and the currently active community around Eclipse

EMF [WSNW07, WSGT09]. The project was developed by the *Distributed Object Computing (DOC)* group at the University of Vanderbilt's ISIS in conjunction with other companies, like Siemens, IBM and PrismTech. Key features of GEMS are comprehensive tool support for the rapid creation of visual model editors as well as constraint-based modeling guidance [WSNW06].

MetaEdit+ was first released in 1995 as a descendant of the earlier version MetaEdit [KT08, p. 391]. The modeling workbench is delivered by the company MetaCase [Met09a], which started as a university spin-off in the end of the 1980s. MetaEdit+ is based on the meta-metamodel *GOPPRR (Graph-Object-Property-Port-Role-Relationship)* [SKM96][Met09b, sect. 1.1] and provides an all-in-one solution for rich and easy-to-use support for metamodeling, visual modeling editor creation and code generator specification. The key principle of MetaEdit+ is to cover the entire software development process from language design to executable applications. This can be achieved due to the fact that companies commonly need to serve only very specific domains and don't need to provide solutions for a broad range of domains [Tol06, p. 21].

OSLO is an upcoming project of the Microsoft Corporation that shall provide interesting modeling capabilitites for software development around and beyond Microsoft technologies. OSLO consists of different parts, such as *M*, *Repository* and *Quadrant*. The core specification language M (or *MGrammar*) can be considered as the meta-metamodel of OSLO, but does not strictly adhere to the OMG's metadata architecture [You09]. M allows to specify languages, instances and schemas in parallel. Together with enabling tools, such as Intellipad [Cor09a], OSLO allows to easily provide textual DSLs together with parsers and interpreters. Currently "OSLO" is still a code name, because the project is under development and not yet available.

2.1.3 Platform Abstraction and Model Application

The abstraction process in MDSD comes to an end when the information specified in abstract models is to be transformed into executable programs. In this section, we explain important terms around this transformation process. We define what *platform* in MDSD actually means, clarify on technological granularity in models, and elaborate on *generation from* and *interpretation of* modeled information.

2.1.3.1 What is a Platform?

One exceptional advantage of the model-driven paradigm is the abstraction not only from implementation details but also from platform-specific architectures. Following the generative approach, as already introduced as *Generative Programming (GP)* [CE00], once created software specifications may be applied to arbitrary runtime platforms. This way, error-safety and platform-independence increase both short-term productivity through a higher value of primary software artifacts and long-term productivity through a lower rate at which primary artifacts become obsolete [AK03].

However, in context of the tremendous diversity of available technologies, frameworks, libraries and execution environments, it remains unclear what the term *platform* actually means. Within this thesis we define the term as follows:

Definition Platform: A platform comprises a set of constraints for the execution of a software application. These constraints depend on the type of application and might comprise operating systems (OSs), execution environments, such as virtual machines [Bat05] or application frameworks [JBo08], runtime libraries or context conditions, like an available internet connection.

2.1.3.2 MDA Abstraction Layers

Recall from Section 2.1.2.1 that models may describe various information, including real world entities and technological system knowledge. To facilitate a maximum of abstraction in software development in order to fully utilize the possibilities of MDSD, it is even *necessary* to employ several models for various kinds of information together in integration. This was also witnessed by the OMG and documented in its specification of the *Model-Driven Architecture (MDA)* [MDA03]. MDA defines a specific approach to system development that involves particular elements in a dedicated process. This is in contrast with MDSD, which simply summarizes technologies to automatically produce executable programs from formal models, without properly specifying a particular process [SV06, p. 11].

The MDA specification defines several levels of abstraction, whereby *abstraction* means "the process of suppressing selected detail to establish a simplified model" [MDA03, p. 2]. The different levels consider a system from particular viewpoints. In descending order of abstraction, a system is described in terms of *Computation Independent Model (CIM)*, *Platform Independent Model (PIM)* and *Platform Specific Model (PSM)*. A CIM, or domain model, specifies the environment of and the requirements for a system and does not contain any information about its structure. A PIM describes the structure of a system without stating details that are specific to particular target platforms. A PSM is a platform specific view on a system, i.e., a PIM enriched with platform specific details. In addition to the different abstraction layers, the MDA specifies the process of transformation between them. Here, PIMs are enriched with additional *marks* to identify model elements that shall be mapped to particular PSM constructs [MDA03, p. 3-3]. These *marked PIMs* can then be transformed into PSMs accordingly. Figure 2.4 illustrates the MDA architecture.

Thus, in MDA the very same information is considered from different viewpoints. Hence, aligned to the four layer metadata architecture, the information modeled for each viewpoint has its own MOF metadata architecture stack, as depicted in Figure 2.4.

2.1.3.3 Generation and Interpretation

As explained above, MDSD aims at the simplification of software development by means of modeling technologies. To this end, it is obviously not sufficient to only model software systems in an abstract language, but rather to use these models as input for further processing steps. Only then is it possible to end up in running software applications. Two common ways to process software models are direct *interpretation* or the *generation* of programming code from specified information [CE00] (cf. Section 2.1.2.4). During interpretation, the contents of input models are directly translated into imperative commands of a particular target language. Instead, when generating programming code, software models are used as input for appropriate generator engines. Exemplary model to text transformation engines are the generator framework *openArchitectureWare (oAW)* [ot09] and MOFScript [Old06]. Figure 2.5 illustrates the relation between these two model application possibilities.

Both methods to utilize the information contained in models have their own advantages and disadvantages. Code generation, for example, does not impose any additional functional require-

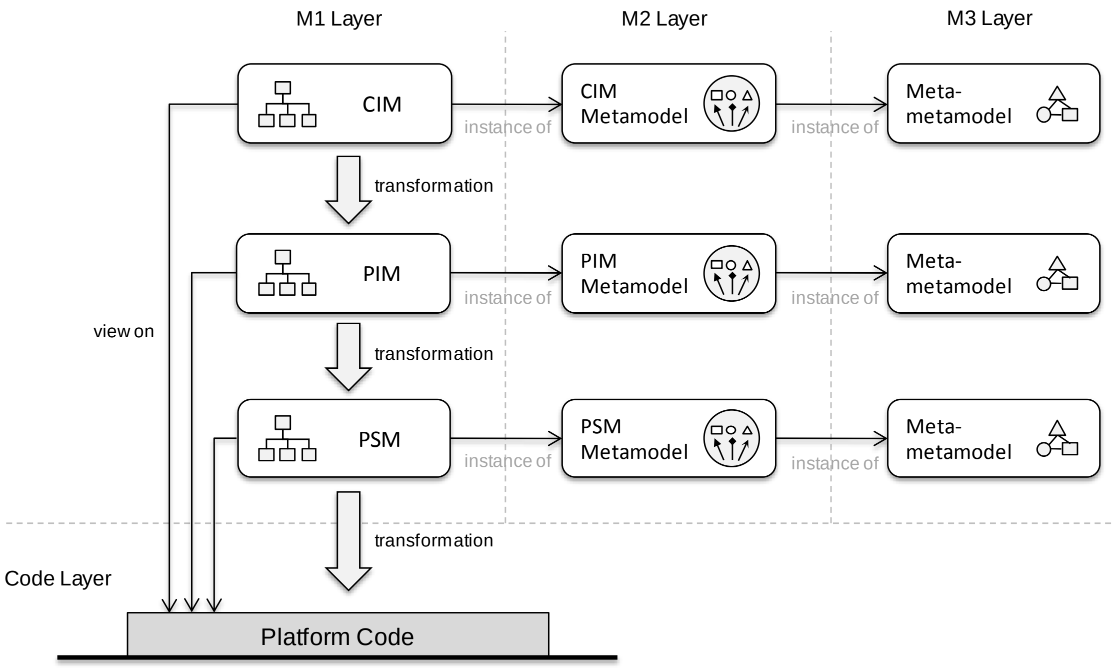


Figure 2.4: *MDA abstraction layers aligned to MOF metadata architecture*

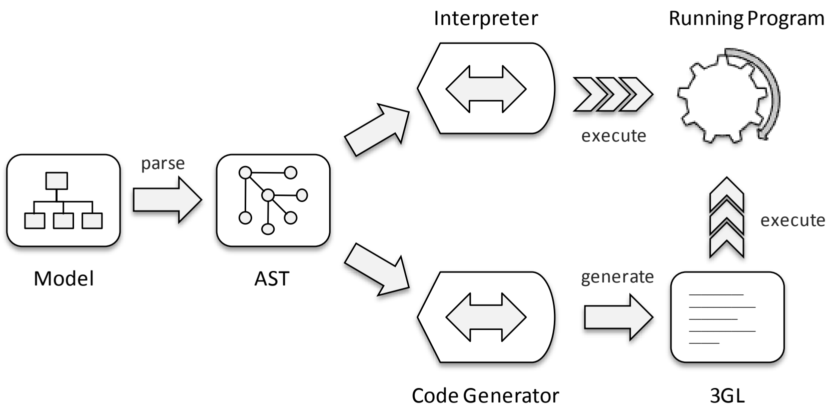


Figure 2.5: *Relation between generation and interpretation*

ments to the target platform, because generated code is entirely independent from the framework that produced this code. Additionally, code generation enables developers to provide precisely tailored assets in order to achieve a maximum of performance, e.g., in embedded domains. Interpreters, on the other hand, have a clear advantage with its late binding time. Because an interpreter *directly* reads and executes models, even late changes to these models can be incorporated at runtime. A preliminary compilation step is not necessary. Of course, this also involves the risk to end up in false system states, in case an erroneous model is provided as input for the interpreting engine. Additionally, the platform (cf. above) needs to provide necessary runtime facilities for the interpreter.

2.2 Domain-Specific Modeling

2.2.1 Overview

The paradigm of model-driven software development is rather young. Around the turn of the millennium [Coo00] the topic began to pick up speed which increased with important specifications (cf. Section 2.1.2.3) and currently reaches a peak with the recent wave of modeling workbenches (see Section 2.1.2.5). As introduced above, the MDSD paradigm simply generalizes the historically grown abstraction process in software engineering. Much older, in this respect, is the trend towards domain specialization. Already in the mid of the 20th century, the abstraction process has led to very specific languages that are appropriate for only very limited application contexts. Prominent examples are the *Structured Query Language (SQL)*, MATLAB [Pal00], and SAP's *Advanced Business Application Programming (ABAP)* [Kel05] language, for the domain of data interaction, or the *Hypertext Markup Language (HTML)* and *Cascading Style Sheets (CSS)*, for the web development domain. Such languages are called *domain-specific languages (DSLs)* (cf. next section).

The zero-tolerance specialization to particular domains renders DSLs as efficient means to development for experts in these domains. Yet, the MDSD community only lately witnessed the advantages of DSLs for model-driven scenarios [WK06]. In the context of MDSD, DSLs are often called *domain-specific modeling languages (DSMLs)*.[1] Today, several modeling workbenches enable developers to create their own languages including editors, transformation specifications, and code generators. The process of creating models based on these languages is called *domain-specific modeling* which yields *domain-specific models (DSMs)* [SV06, p. 71].

All in all, domain-specific modeling aims at describing software not only on a higher level of abstraction, like generally intended by MDSD, but rather specializing these abstractions toward particular problem domains. This way, software development is not restricted to high-skilled developers anymore but enables also technically less-educated domain experts to deliver software implementations. Through a clear tailoring of notations and modeling primitives to a particular domain, a DSL narrows the developers mental model and facilitates precise, fast and efficient development without tedious ramp-up phases.

2.2.2 Domain-Specific Languages

Domain-specific languages (DSLs) are languages that are dedicated to a particular application domain by providing special notations and constructs tailored toward that domain. Hence, such languages trade generality for "substantial gains in expressiveness and ease of use" [MHS05] in a limited context. Domain-specific languages can be of both graphical and textual nature, according to their purpose and problem domain. In this respect, graphical DSLs are called *domain-specific visual modeling languages (DSVMLs)* [GPvS02]. As introduced above, DSLs are rather old and usually classified as *fourth generation languages (4GLs)* [MHS05].

A domain-specific language is composed of an appropriate metamodel, which defines its abstract syntax and static semantics (cf. Section 2.1.2.2), and a set of textual and/or graphical notations, the *concrete syntax*, that is used to create "sentences" or models of the language [SV06]. Cook divides DSLs according to the kind of domain they are developed for [Coo06]. To this end, he classifies vertical, business domains and horizontal, technical domains. The domain orientation, in this respect, indicates the bandwidth of applicability of a domain in relation to other domains. For example, the horizontal, technical domain of *user interfaces (UIs)* applies to

[1] Within the remainder of this thesis, we continue to use the term DSL instead of DSML.

various vertical application domains, because in each of these domains the UI is very important. Consider Figure 2.6 to clarify on the relation between the two different domain types.

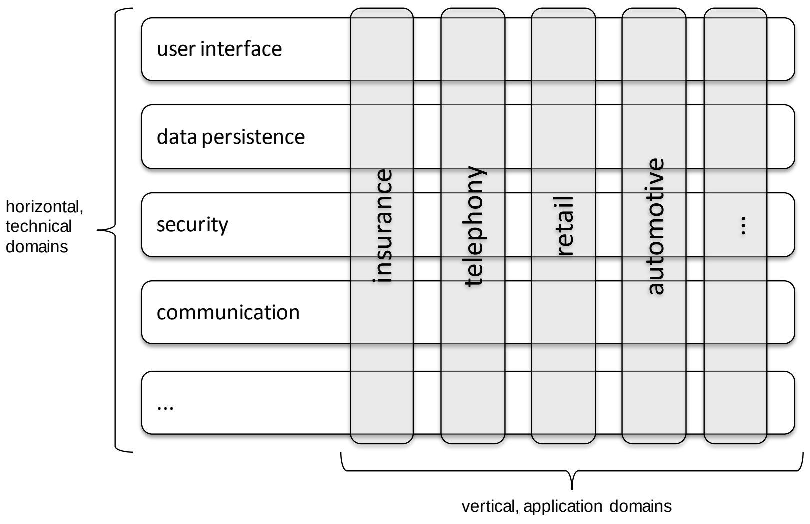


Figure 2.6: *Business domains and technical domains in relation*

Specialized to particular domains, DSLs are much more expressive and easy to use in the limited context. They offer substantial gains in separation of concerns and development efficiency, on the one hand, while reducing the amount of necessary technical expertise for development on the other hand. However, the development of a DSL is rather expensive. Besides the development of the language description via its metamodel and concrete notations, appropriate tools must be developed that provide convenient modeling support. Additionally, a proper balance between requirements for and capabilities of a DSL must be found in order to ensure that the language can be reused in different contexts. Moreover, even if it has been achieved to define a well suited, self-contained, and reusable domain-specific language, it remains nearly impossible to reuse its *implementation* (a code generator or an interpreter, see Section 2.1.3.3) in a different context.

2.2.3 Development with Multiple DSLs

Development with one single DSL raises the efficiency in software engineering, as described above. But, software systems consists of a multitude of different application parts such as the user interface or the persistency layer. DSLs are tailored to single application domains which causes their actual benefits in the end. Hence, to cover the specification of entire software systems and efficiently handle each of the involved domains, it becomes necessary to offer multiple views on such systems. The solution is the employment of multiple DSLs in combination.

The potential of multi-DSL development was already witnessed quite early. The first workings in this context include the publications of Finkelstein et. al, who introduced a "[...] framework for integrating multiple perspectives in system development" [FKN+92]. The authors coined the notion of "viewpoints" and elaborated on the challenges of multi-DSL development [FGH+94]. Today, many application frameworks and tool suites provide platforms for software development with various DSLs [GS04, PC05, WK06, Fou09d, YH07]. However, while the prevalent understanding of multi-DSL development agrees on the requirement of employing multiple languages, it often remains unclear how the distinct languages inter-operate to form a complete system, which finally implements an executable application. To clarify this, we review two types of scenarios for development with multiple DSLs, namely interpretive and generative systems, in the following two sections.

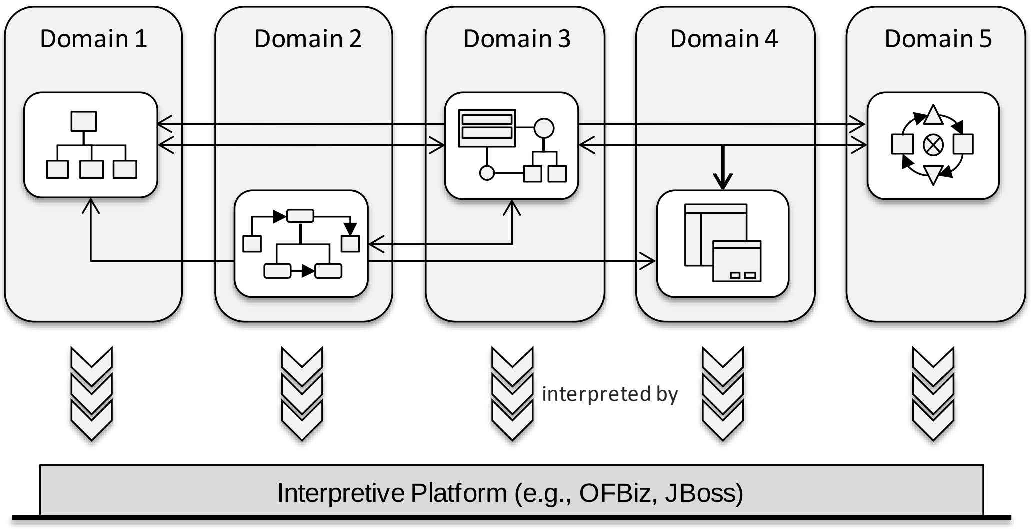


Figure 2.7: *Multi-DSL development with interpretive systems*

2.2.3.1 Interpretive Systems

One common approach to multi-DSL development predominantly stems from the area of web application engineering. Here, huge managed application servers necessitate the employment of complex configuration files and other declarative assets, such as persistence mapping descriptions, navigation rules through web pages or user interface dialogs. The different specifications have to be connected to the actual business logic and data entities, commonly implemented with a general purpose programming language. The whole integration task is accomplished by the underlying web application frameworks and application servers respectively. Examples for such platforms are the *JBoss Enterprise Application Platform* [JBo08], the *Apache Open for Business Project (OFBiz)* [Fou09d] or the *Spring Framework* [Joh07].

Such systems are interpretive, i.e., the different software artifacts are parsed by the runtime system and interpreted by executing appropriate commands on the target platform (cf. Section 2.1.3.3). For example, often navigation rules control the presentation order of web pages depending on user input or system events in web application frameworks [JBP08]. Such rules usually contain identifiers for corresponding pages. The underlying runtime platform, which is aware of the structure and contents of navigation rules on the one hand, and the available web pages on the other hand, automatically handles the presentation of particular pages that belong to the identifiers specified in a rule. Hence, because the knowledge about particular DSLs is encoded in the runtime platform, connections between software parts can be reduced to simple, *string-based references*. An explicit treatment of the interplay and interdependencies between the different languages and their models is not necessary.

Figure 2.7 illustrates such interpretive scenarios. Depicted is a development scenario, which is composed of five different domains. Each domain contains one[2] domain-specific model that maintains references to one or more other domains. As shown, all DSMs are entirely interpreted by the underlying execution platform without any mediating structures.

[2] Although the illustration shows only one DSM per domain, this is of course no obligation and done for brevity reasons only.

2.2.3.2 Generative Systems

An alternative approach to implement multi-DSL scenarios is the employment of code generators. A code generator processes binding specifications that map concepts of a modeling language to particular code assets (see Section 2.1.3.3). In contrast to interpretive systems, multi-DSL scenarios in generative systems do not require one single platform that is aware of all employed DSLs. Instead, code generators for each language must produce executable runtime code, appropriate configuration files or, more generally speaking, assets that can be processed by the corresponding platform.

As an example assume a platform based on a GPL, like the *Java 2 Standard Edition* [Blo08], equipped with various code libraries for different concerns such as the persistent data storage or the UI presentation. Corresponding DSLs for data structures and the user interface would have to generate appropriate platform-specific code that suffices the underlying libraries. Even if proper references between different DSMs would be maintained, e.g. with string-based values (cf. section above), the information about the meaning of these references is not implemented in the employed platform libraries. To bridge the gap between assumptions about the interplay of DSMs on the modeling layer and the actual implementing code that is produced by code generators additional integration structures need to be provided. In generative scenarios this is usually done by adjusting generator templates for one single domain to import and process information of adjacent, connected domains. Hence, the knowledge about the interplay of different domains is implicitly encoded in generator templates.

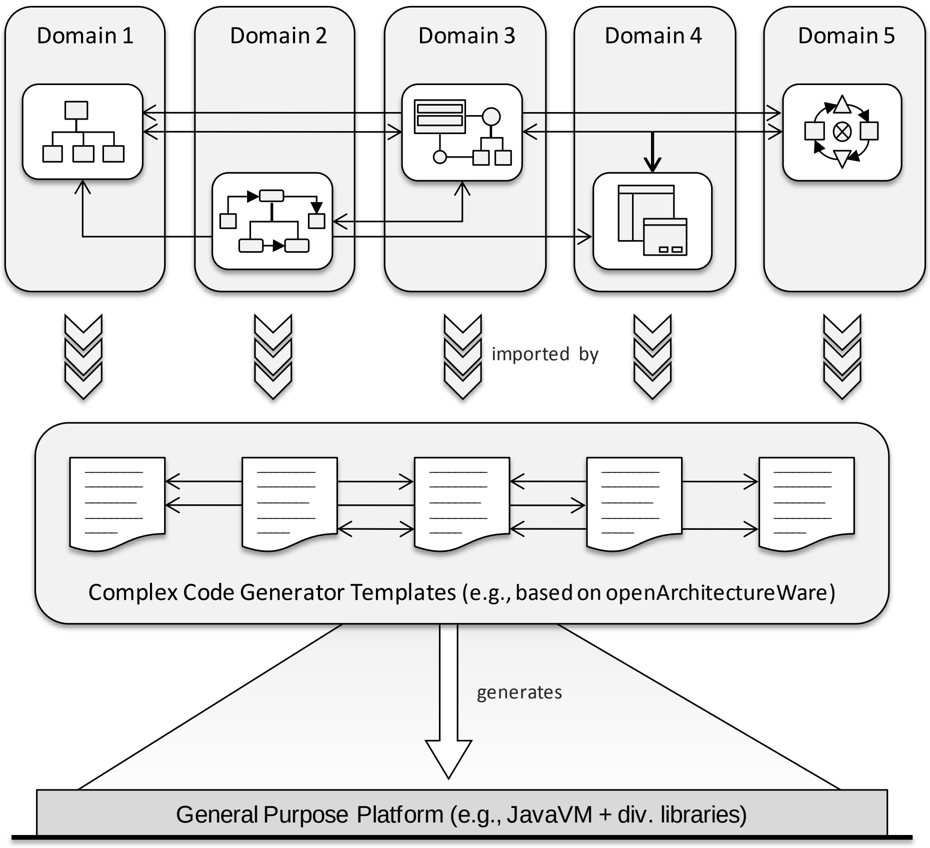

Domain 1
Domain 2
Domain 3
Domain 4
Domain 5
imported by
Complex Code Generator Templates (e.g., based on openArchitectureWare)
generates
General Purpose Platform (e.g., JavaVM + div. libraries)

Figure 2.8: *Multi-DSL development with generative systems*

Figure 2.8 illustrates multi-DSL development with generative systems. In difference to Figure 2.7, the interconnected DSMs of the different domain are no longer processed directly, as it is the case in interpretive systems. Instead, the knowledge about language dependencies and interaction is implicitly contained in code generators which need to produce valid platfom-specific programming code. Here, the complexity of artifact interplay has to be expressed with means amenable by the targeted platform. This is why the corresponding generator templates can evolve to very complex structures.

2.3 Software Product Line Enginering

2.3.1 Overview

In parallel to abstraction and domain specialization, an adjacent community considers the challenge to implement and maintain variability in software. In 1976, Parnas coined the notion of *program families* that is a collection of software applications that share common base assets to a large extent [Par76]. On top of this software core, each family member adds individual extensions as variable assets to form a distinct product in a stepwise refinement process. Parnas argued for the advantages of studying the commonalities of applications in program families.

Today, the term program or system-family is significantly used in the area of *Software Product Line Engineering (SPLE)* which is the paradigm for product line development. The clear focus of SPLE lies on developing software applications from reusable core assets rather than creating them from scratch [KLD02, LKL02]. In this respect, a software product line denotes “a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment” [SEI08]. Hence, software product lines have always a clear business intention in addition to the technical notion of program families. Not only an analysis of the problem domain but rather a market analysis forms the foundation for products and their features [LKL02].

2.3.2 Development Elements

The product line engineering process distinguishes between two major elements, namely problem space and solution space. A problem space describes entities that pose challenges or requirements to be solved by some solution space. Before we elaborate on the two terms in more detail, it is important to mention that both spaces are always directly related to each other. Particular problem and solution spaces form pairs, whereby the solution space of a problem space itself can perfectly well act as problem space for another solution space. In Figure 2.9 the relationship between both product line development elements is illustrated. The illustration shows two problem/solution space pairs, where each contains a range of artifacts. The example shows a generative scenario, where (from left to right) a feature model (see next section) is “solved by” some abstract technical models, which themselves are used as problem space for code generators that produce actual programming code in the end.

2.3.2.1 Problem Space

The problem space in SPLE denotes “a set of domain-specific abstractions that can be used to specify the desired system-family member” [Cza05]. Hence, it comprises tools to enable engineers to specify their needs in a way that is natural for their domain. Because problem and solution space are always aligned relative to each other, it is hard to specify which assets exactly belong to *the* problem space. Nevertheless, product line development starts like other software projects with high-level analysis tasks. A prominent example for typical problem space models are feature models, as explained in the following.

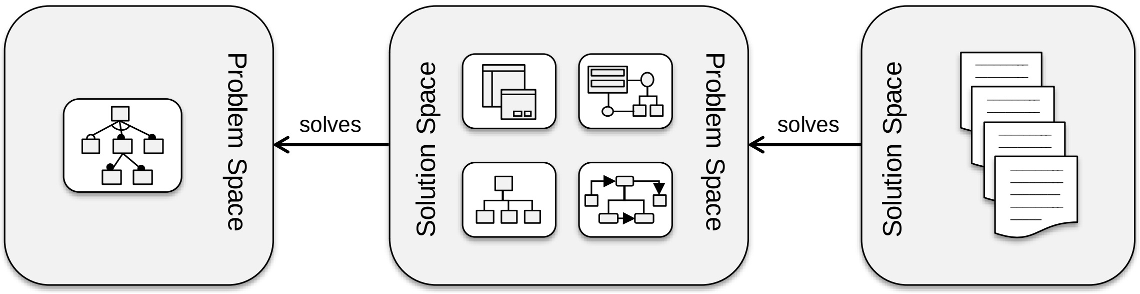


Figure 2.9: *Problem space, solution space and their relation*

Feature-Oriented Domain Analysis

In 1990, Kang et al. presented an approach to analyze program families for these commonalities and their corresponding variable parts, the *Feature-Oriented Domain Analysis (FODA)* [KCH+90]. The FODA approach is led by the idea to describe the capabilities of software in form of product *features*, which denote prominent and distinctive user-visible characteristics of a system [LKL02]. To this end, commonalities and variabilities in entire software families are specified in *feature models* that reflect the "end-user's (and customer's) understanding of the general capabilities of applications in a domain" [KCH+90]. With feature modeling a domain can be analysed from a customer-centric point of view. This is why feature models do not specify an entire domain as it is, but rather concentrate on market relevant aspects of products in a domain (cf. next section). Hence, feature models are market-specific slices of a domain of interest, which result from careful analysis of market requirements.

The original FODA notation distinguishes between *mandatory*, *optional* and *alternative* features. To this end, Kang et al. proposed to aggregate features in *feature groups* that manage the cardinality of their sub features accordingly. In [CHE05], Czarnecki et al. proposed an extension for the original FODA notation by introducing cardinalities and references to improve clarity and support for modularization. Figure 2.10 shows an exemplary feature model.

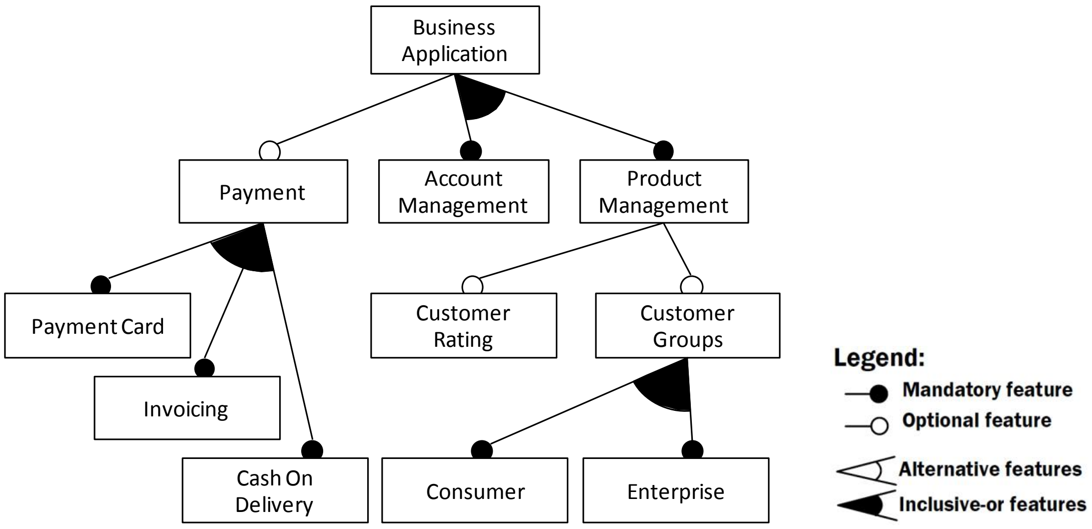


Figure 2.10: *Example for a functional feature model*

Also in [CHE05], Czarnecki et al. emphasize that feature models are not only "user-visible" product properties, as initially proposed by Kang et al. in their original definition. Instead, the authors highlight that features should be specific to and allowed for various stakeholders, such as customers, analysts, developers and so forth. In this thesis, we go in line with the latter definition but clearly emphasize the market specific impact of features. The value of features arises through their reflection of market requirements. Of course these requirements must not only be of functional, non-technical nature, but can rather also specify technical variability of a product line that is relevant to corresponding stakeholders, such as architects or system administrators. Nevertheless, a feature model *should not* reflect the entire variability of a product line, including variation points that are not relevant to potential end users. This would entail that the complete system design is reflected in feature models, which is out of scope in feature modeling.

2.3.2.2 Solution Space

A software product line's solution space "consists of implementation-oriented abstractions, which can be instantiated to create implementations of the specifications expressed using the domain-specific abstractions from the problem space" [Cza05]. As introduced above, problem and solution space are aligned relative to each other. Complex product line scenarios may involve a multi-layered stack of problem/solution space pairs. As with problem space assets, this makes it difficult to constitute what types of assets exactly belong the solution space, because it always depends on the problem space that declares the point of view from which a solution space is considered.

Nevertheless, the common understanding of solution space assets highlights its implementation-oriented focus, as cited above. Thus, at least all assets that belong to the actual technical realization of a product line form its solution space. This includes analysis and design models, domain-specific languages, code generators or configuration artifacts, just to name a few.

2.3.3 Development Stages

Software product line engineering comprises two major stages, namely *domain engineering (DE)* and *application engineering (AE)* [Cza05]. Thus, SPLE shares its process model with other system-family approaches, such as *Generative Software Development* [CE00, Cza05]. The domain engineering stage, as "development for reuse" [Cza05], is understood as the phase of creating arbitrary assets that belong to a particular application domain. During application engineering, as "development with reuse" [Cza05], the previously created assets are used to build concrete products. Figure 2.11 illustrates the relationship between domain and application engineering.

2.3.3.1 Domain Engineering

During the domain engineering phase in product line development, so-called *core assets* are created that form the basis for the implementation of actual products. While sometimes simply referred to as *asset development* [KLD02], the term domain engineering reflects better that this development phase is comprised of various activities on most diverse abstraction layers. The DE phase involves market analysis activities, such as the creation of *marketing and product plans (MPP)*, feature models and requirement analysis, as well as the implementation of the core product line architecture, where reusable components are provided. Hence, during this phase essential assets for both problem and solution space are developed.

2.3.3.2 Application Engineering

The application engineering phase denotes the point in development when all assets that were previously developed during domain engineering are composed to an executable end product.

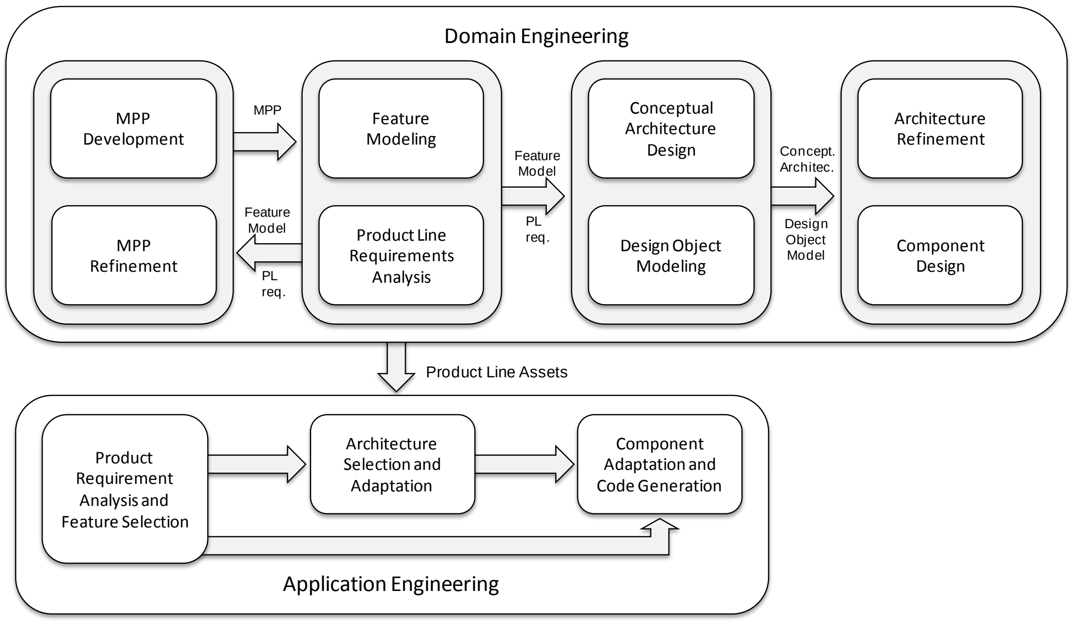


Figure 2.11: *Domain engineering, application engineering and their relation (adapted [KLD02])*

The AE phase is performed on the basis of a particular product variant, which is specified by a so-called *variant model* [HL06], a valid instance of a product line's feature model. The term *valid*, in this respect, indicates that the variant model suffices the semantic obligations of the underlying feature model, including implicit metamodel constraints and optional explicit restrictions between features. The application engineering phase comprises the selection of corresponding solution space artifacts, their proper composition and instantiation or configuration. These steps are performed either automatically, based on an explicit binding between features and their implementing artifacts, or manually, with an implicit mapping. In line with others [BH01], we summarize these activites under the term *product instantiation*.

Figure 2.12 illustrates the product instantiation process. The upper part of the illustration depicts a valid variant model as instance of the feature model in Figure 2.10. Each feature in the choosen variant includes the mentioned mapping to partiular artifacts. Commonly, the assets of a product line can be divided into a minimal fixed application core and various feature-centric code assets [VG07]. The product instantiation process now leads from the fixed core to a shippable product, while merging each feature's assets into core structures.

2.4 Ontologies and the Semantic Web

2.4.1 Ontologies

2.4.1.1 Important Terms

In literature, many publications can be found that discuss the terminology around logical foundations of artifical intelligence. Thus, a whole bunch of papers comprehensively define important terms in that area, such as *conceptualization*, *ontology*, *knowledgebase* or *ontological commitment* [GN87, Gru95, GG95, Gui05, Obe06]. This is why we limit our introduction of selected terms relevant for the work in this thesis to a concise summary and refer to related work for further details.

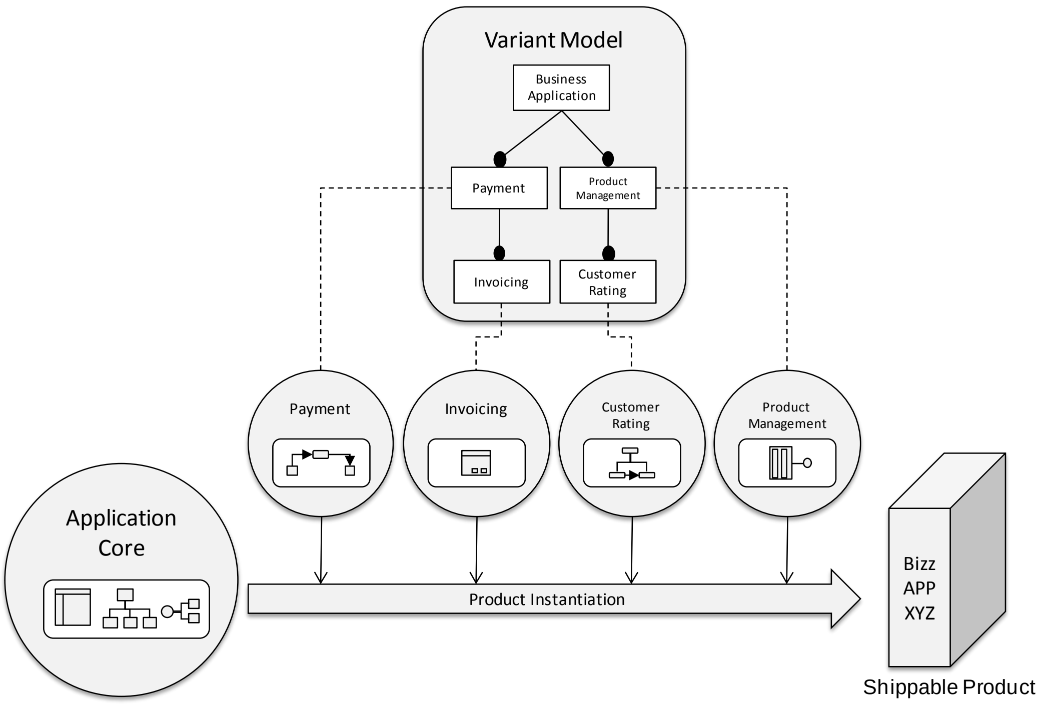


Figure 2.12: *Product instantiation during application engineering*

Conceptualization is a term that was actively discussed by the research community [GN87, Gru95, GG95]. In short, a conceptualization describes a dedicated *universe of discourse* [GN87, p. 10], i.e., a particular domain of interest. This comprises concepts of that domain D and relationsships between these concepts R. Hence, a conceptualization C is defined by the tuple (D, R). Later, Guarino et al. [GG95, Gua98] pointed out that the meaning of relations may differ in different worlds, which led to a definition of conceptualization with respect to such worlds W, expressed in the triple $C = (D, W, \mathfrak{R})$. For further information and additional details, the reader is referred to [GN87], [GG95] and [Gua98].

Ontology has different meanings, where a major difference is drawn between "Ontology" (with capital O) and "ontology" (with lower case o). The former denote a philosophical discipline that describes Ontology as the science of being, firstly mentioned by Aristotle in his *Metaphysics* book. The second meaning understands the term as "an explicit specification of a conceptualization" [Gru95]. For a comprehensive explanation of the term ontology, the reader is referred to [Gui05].

Ontological Commitment is explained by Guarino as "a partial semantic account of the intended conceptualization of a logical theory" [GG95, p. 6]. Guizzardi discusses this term in connection with languages and metamodels in [Gui07]. In this publication, the ontological commitment is defined as the set of ontological entities the language primitives *commit* to the existence of.

TBox stands for *terminological box*. It "stores a set of universally quantified assertions (inclusion assertions) stating general properties of concepts and roles." [GL96]. Therefore, the TBox encapsulates all type-level axioms that state generally valid semantic structures for a particular domain of discourse — the *terminology* of that domain [BCM$^+$07, p. 50].

ABox stands for *assertional box* and "comprises assertions on individual objects (instance assertions)." [GL96]. Hence, it contains instance-level assertions that describe knowledge about concrete entities in terms of the vocabulary given by its terminology [BCM$^+$07, p. 50].

In literature, a *knowledgebase (KB)* encapsulates knowledge about a dedicated domain. In contrast to the modeling community in software engineering, there is often no clear distinction between type-level and instance-level assertions. While in MDSD a model on layer M1 (cf. Section 2.1.2.3) is clearly separated from its defining metamodel on layer M2 a knowledgebase might comprise both TBox assertions in form of an ontology and ABox assertions about concrete entities [BCM+07, p. 50].

In this thesis, we follow the explanation of Guizzardi in [Gui05], who summarized that a *generic knowledgbase* is comprised of an *ontology* “containing situation-independent information” and a *core knowledgebase* “containing situation-dependent information” [Gui05, p. 67]. To this end, each time where we use the term “knowledgebase” or KB, we mean what Guizzardi referred to as “core knowledgebase”, i.e., a container that encapsulates ABox assertions only. Likewise, when mentioning an “ontology”, we mean a container that encapsulates TBox assertions only. Hence, we try to align the relationship between TBox and ABox directly to the relationship between the layers M2 and M1 in MDSD.

2.4.1.2 Ontology Hierarchy

There are different types of ontologies as well as different ways to classify them. Available classifications distinguish ontologies according to specificity, purpose, application possibilities, expressiveness or applicable inference mechanisms [NH97][OB06, p. 44]. In [Brä07b], Bräuer developed a simplified classification which summarizes the most important ontology types and arranges them according to their degree of domain-specificity. Figure 2.13 illustrates the result.

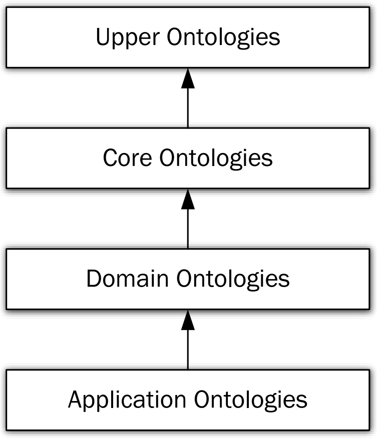


Figure 2.13: *Simplified ontology classification (from [Brä07b])*

The top of the hierarchy consists of *upper ontologies* (in literature often referred to as *foundational ontologies*), which contain very general concepts that are applicable in most — if not all — domains. *Core ontologies* define concepts shared across a number of domains that are related to each other. For example, an ontology for general business purposes might contain concepts like customer or company, which might be relevant for several business areas, such as Customer Relationship Management (CRM) or Product Lifecycle Management (PLM). A *domain ontology* comprises semantic structures tied to one particular domain. A PLM ontology would introduce appropriate concepts for product lifecycle states that are not relevant in the CRM domain. *Application ontologies* are specialized versions of domain ontologies, defining different forms of the same domain. For instance, in the domain of workflow management exist a multitude of different variations that often alter in modeling granularity.

2.4.2 Semantic Web

In the last decade, the internet evolved to a main source of information for the worldwide community of its users. Data can be requested through text-based queries in dedicated search engines that crawl the internet along hypertext links between various websites. Now, this kind of use is rather restricted, because it is limited to pure syntactical queries about content available in textual form. Users are obliged to inventively combine several search strings, always trying to leverage the search algorithm of the corresponding search engine. Significant queries based on the meaning of published information are still not state of the art.

The need for appropriate mechanisms to allow semantic queries was recognized already in the year 2000 by Tim Berners-Lee [BL00]. Berners-Lee et al. coined the vision of the *Semantic Web* [BLHL01] which core idea is to furnish existing information available in the internet with additional well-formalized metadata, amenable to appropriate inference engines. This way, content provided on the internet should evolve from solely *machine-readable* to *machine-understandable* information. To this end, Berners-Lee presented the Semantic Web architecture, involving several formats and technologies as depicted in Figure 2.14.

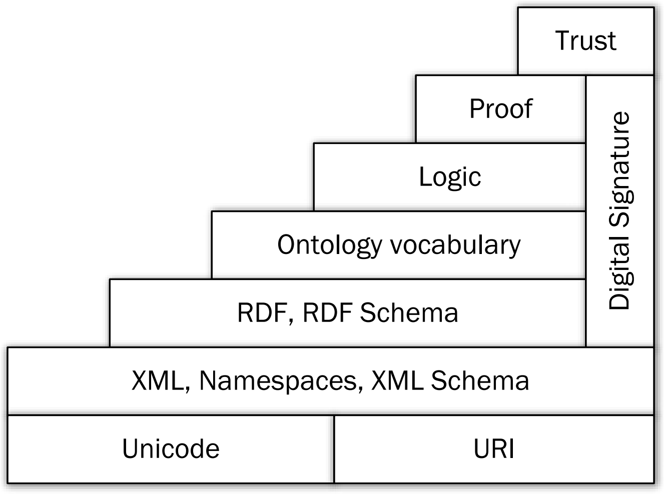


Figure 2.14: *The Semantic Web Architecture (from [BL00])*

The architecture is based on well established standards, such as Unicode, *Unified Resource Identifier (URI)* [BLFM05], the *Extensible Markup Language (XML)* [BPSM⁺08], *XML Namespaces* [TBHLT06] and *XML Schema* [FW04]. On top of these standards, several extensions augment semantic expressiveness in dedicated layers. This comprises particular formats as well as additional layers that encapsulate the logical inference capabilities, as summarized in the next sections.

2.4.2.1 Formats and Languages

The W3C standardized a number of formats and languages to reach the envisioned goal of augmenting syntactical structures of the internet and formalizing machine-understandable semantics, respectively. In the following we introduce the most important technologies for this work.

The Resource Description Framework (RDF) is the most basic language for representing information about resources on the internet, as also indicated with the RDF, RDF Schema layer in Figure 2.14. RDF follows a straightforward concept for resource annotation. Simple facts are expressed with triples of *subject*, *predicate* and *object* [MM04], which facilitate statements like “this thesis - is written by - Henrik”. A set of triples forms an RDF graph, where nodes are either subject or object and predicates denote a relationship [KC04]. RDF nodes are URIs, URI references, blank nodes or literals (cf. next section), while literals must be used as objects only. Predicates are URI references exclusively. RDF can be serialized in different notations, such as *Notation 3 (N3)* or the RDF XML format [Bec04].

RDF Schema (RDFS) extends RDF with additional constructs to establish RDF classification hierarchies [BG04]. The language introduces concepts to define classes and properties (cf. next section) as well as class and property hierarchies with corresponding properties, such as `rdfs:subClassOf` or `rdfs:subPropertyOf`. This facilitates the creation of simple ontologies.

The Web Ontology Language (OWL) is a revised version of DAML+OIL [CvHH^{+}01] and provides further vocabulary together with formal semantics that allows to build much more expressive ontologies than RDFS does [SWM04]. This includes advanced vocabulary for classes and properties, such as cardinality restrictions, equality relations or property symmetry, that enable machines to perform powerful reasoning tasks. The language is divided into three different sub languages: *OWL Lite*, *OWL DL* and *OWL Full*, with increasing expressiveness in enumerated order. OWL Lite as the least expressive language comes with basic features to establish classification hierarchies with a minimal set of constraints. OWL DL, named after its logical foundations of *Description Logics (DL)* (see Section 2.4.3), provides a maximum of expressiveness while remaining computational complete and decidable, i.e., every language construct is computable and computation will finish in finite time [PSHH04]. OWL Full even enriches the expressiveness of OWL DL, but at the price of computational guarantees.

2.4.2.2 Entities and Visualization

All of the above introduced languages allow to create semantic structures based on diverse foundational entities. This section provides an overview of the most important entities and explains how they are related to each other. Corresponding explanations are mainly taken from [MvH04]. In addition, we present the diagrammatic syntax that will be used to visualize ontologies in the remainder of this thesis.

Classes represent groups of individuals that belong together because they share particular properties. As indicated above, classes can be organized hierarchically through the `rdfs:subClassOf`. In addition, OWL defines `owl:Thing`, which is the common super class for all classes and base class for all individuals, respectively.

Properties are used to establish relationships between individuals. Corresponding definitions are created with `rdf:Property`. Properties can be assigned to particular classes with `rdfs:domain` and constrained with data type restrictions through `rdfs:range`. In OWL, one additionally distinguishes between `owl:ObjectProperty` and `owl:DatatypeProperty` to restrict property ranges to individuals of defined classes and RDF literals and XML Schema datatypes, respectively.

Individuals are instances of classes. Properties, either previously defined or not, put different individuals into relation.

Literals are lexical strings to express values like numbers or dates. One distinguishes between plain literals, i.e., untyped strings that may be equipped with an optional language tag, and typed literals that contain a datatype URI, which usually denotes an XML Schema datatype.

In Figure 2.15 we present a small example to illustrate the usage of above explained entities and languages, and introduce their diagrammatic visualization, which in principle is a slightly differed variant of the RDF graph notation introduced in [KC04].

The exemplary ontology stems from the domain of text documents and describes in this special case the organization of PhD theses. As shown, ellipses depict ontology classes, trapezoids represent properties and diamonds are individuals. Heritage relationships, such as defined by `rdfs:subClassOf` or `rdfs:subPropertyOf`, are represented by solid lines with bigger open arrows. Instantiated properties, i.e., individuals that have allocated particular properties with certain values, are represented by solid lines with small barbed arrows.

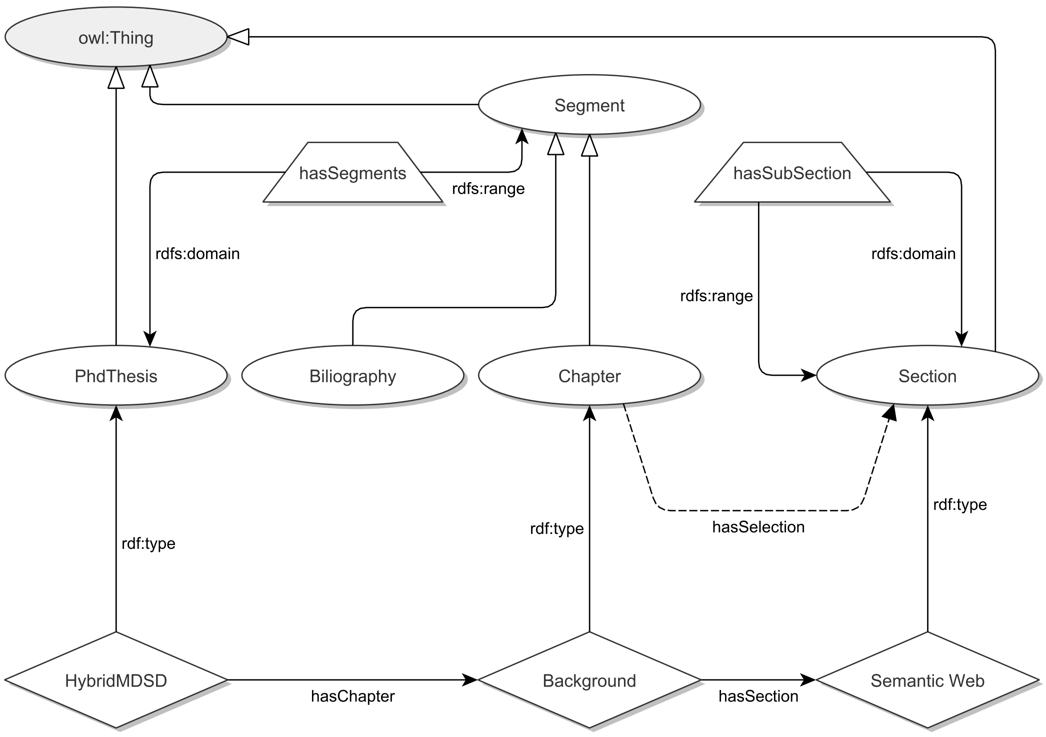


Figure 2.15: *An exemplary ontology for documents*

The depicted OWL ontology defines the structure of doctoral theses with appropriate ontology classes, such as PhdThesis, Segement or Section. As described above, each OWL class automatically subclasses the core class owl:Thing, which is differently colored than the other nodes. Such a coloring shall indicate that particular ontology parts are defined in different namespaces and imported, respectively. A thesis is comprised of different segments, as defined by the hasSegments relationship, with rdfs:domain set to PhdThesis and rdfs:range set to Segment. Different segment types are Bibliography and Chapters, while a Chapter might contain Sections and this in turn, might contain additional subsections, as captured by the hasSubSection property. This lower part of Figure 2.15 illustrates several individuals that represent the structure of this thesis down to the actual section. Corresponding instantiation relationships are indicated by the rdf:type properties. Please note the alternative property definition presentation, shown with the hasSection property.

2.4.3 Reasoning, Description Logics, and Rules

Besides facilitating to just capture semantics for dedicated domains, ontologies and Semantic Web languages allow to derive additional, not explicitely modeled information from a given source of knowledge. Based on logical inference and optional rules, sophisticated computational engines are able to process ontologies or knowledgebases and produce unconsidered facts and inferred conclusions. This process is called *reasoning*. In literature, there is a general distinction between (1) *Description Logics (DL)* reasoning and (2) rule reasoning, which is shortly explained in the following subsections.

2.4.3.1 Description Logics Reasoning

Description logics denote a family of knowledge represenation languages, with members such as $\mathcal{ALC}$, $\mathcal{SHOIN}$ and $\mathcal{SHIQ}$. The family members are named after their logical expressiveness, indicated by the letters that occur in their names. In this respect, each letter denotes a particular expressivity. $\mathcal{ALC}$, for instance, consists of $\mathcal{AL}$, which stands for attributive language, and $\mathcal{C}$, which denotes complex concept negation. The languages OWL Lite and OWL DL provide the expressiveness of $\mathcal{SHIF}$ and $\mathcal{SHOIN}$, respectively. For a complete reference of all available expressivity letters, the reader is referred to [BCM+07].

The term *description logics* exposes that in DL the important notions of a domain are given by *descriptions*, i.e., expressions built with unary predicates (*atomic concepts*) and binary predicates (*atomic roles*) [BCM+07]. As examples consider the following two description logic axioms, which establish the concept for a customer as well as a purchase relationship.

$$Customer(x) \tag{2.1}$$
$$purchase(x, y) \tag{2.2}$$

For concept creation, so-called constructors can be used, such as *conjunction* ($\sqcap$ — interpreted as intersections), *negation* ($\neg$ — interpreted as set complement) or *existential restrictions* ($\exists R.C$). Thus, concepts may also be established as illustrated with the next axiom, which defines that a customer is a person who purchased something.

$$Customer(x) \equiv Person(x) \sqcap purchase(x, y)$$

DL systems allow to store terminologies and assertions, i.e., TBox and ABox, and to *reason* about them. The first system that provided profound (description) logics implementation was KL-ONE [BS85], which had an influence on development for the entire language family [BCM+07, p. 19]. Today, there are several systems available for different programming languages, such as FaCT++ [TH09], KAON2 [KAO09] or Pellet [CP09]. Typical reasoning tasks differ between TBox and ABox. Prominent terminology deductions answer questions about *satisfiability*, i.e., whether a concept makes sense or is contradictory, or *subsumption*, i.e., whether one concept is more general than another. Common ABox reasoning tasks deal with questions about the *consistency* of knowledgebases, i.e., whether it has a model, or *instantiation*, i.e., whether an individual suffices a concept from the terminology.

2.4.3.2 Rule Reasoning

Rule reasoning denotes inference drawing on *rule systems*, i.e., a subset of predicated logic which is also referred to as *Horn logic* [AvH04]. Rules consist of two elements, the *head* and the *body*. As example, consider

$$A_1, ..., A_n \rightarrow B$$

where $A_1, ..., A_n$ is the rule's body and B its head. $A_1, ..., A_n, B$ are atomic formulas and the commas are read as conjunctions. The listed rule is interpreted as, if $A_1, ..., A_n$ hold true, then also B is known to be true. Hence, the body $A_1, ..., A_n$ comprises the *premises* of the rule, while B is its *consequence*. Atomic formulas may contain variables. For example,

$$father(X, Y), brother(X, Z) \rightarrow uncle(Z, Y)$$

is valid and bears universal qualification over all employed variables, using $\forall X \forall Y \forall Z$. According to [WGL06] and [AvH04, p. 152], one distinguishes between several kinds of rules:

Deductive Rules denote rules that are used to infer implicitly contained knowledge and are interpreted as, if $A_1, ..., A_n$ is true then B is true as well.

Reactive Rules are those kind of rules that trigger the manipulation of a knowledgebase. They are read as, if $A_1, ..., A_n$ is true, then carry out the action B.

Integrity Rules identify constraints that influence the integrity state of systems and are interpreted as, if $A_1, ..., A_n$ is true, the B *needs* to be true as well. An interesting distinction exists between *alethic* and *deontic* rules. The former express *necessary* logical consequences, while the latter express the need or the *obligation* for particular circumstances in order to reach integrity.

Furthermore, a distinction is made between *monotonic* and *nonmonotonic* rules. Above considered rules can always be applied in a sense that if the body is true, the head is true as well. Nonmonotonic rules, instead, are allowed to employ *negated atomic formulas* in the body or head, which leads to the possibility that particlar rules contradict each other. For example, the two rules

$$p(X) \Rightarrow q(X) \tag{2.3}$$
$$r(X) \Rightarrow \neg q(X) \tag{2.4}$$

are in conflict. Assumed the two facts $p(a)$ and $r(a)$, a reasoning algorithm would conclude neither $q(a)$ nor $\neg q(a)$. As shown in the previous listing, nonmonotonic rules are indicated by a double arrow $\Rightarrow$. Such rules are also called *defeasible* [AvH04, p. 161], because one rule can be defeated by another. One possibility to conclude valid results also from nonmonotonic rules is to introduce rule priorities, as proposed, for instance, in [Ant02].

In recent years, the W3C made several efforts to establish standardizations for rule application in the Semantic Web. As a result, the *Semantic Web Rule Language (SWRL)* [HPSB$^+$04] was proposed as member submission. SWRL extends OWL Lite and OWL DL with sublanguages of the *Rule Markup Language (RuleML)* [Rul09]. This facilitates the definition of Horn-like rules for OWL ontologies. The standard additionally defines an appropriate XML format based on OWL, RDF and RuleML. Today, several applications provide support for SWRL rule reasoning, such as the above mentioned KAON2 or the SWRLTab plugin for the prominent ontology modeling tool Protégé [SMI09]. Besides standardized rule languages, also proprietary systems exist, such as the built-in general purpose rule engine of the Semantic Web framework Jena [Jen09].

Chapter 3

Problem Analysis and Vision Outline

In Chapter 1 and 2, we shortly motivated the work presented in this thesis and set up necessary background information to understand the discussions in the remainder of this document. This chapter provides a detailed analysis of the set of problems that we try to solve. This analysis serves as basis for the targeted solution approach, presented in Chapter 4. To this end, we start this chapter by introducing a motivational example in Section 3.1. The example will be used to illustrate and discuss actual problems and deficiencies in modern software development processes. Based on this detailed problem description a comprehensive vision outline is sketched in Section 3.3. What requirements are necessary to realize this vision is further elaborated in Section 3.4.

3.1 Motivational Example

In this section we introduce a motivational example, which shall illustrate and summarize the challenges with which today's software engineering industry is confronted. Assumed is a software company that develops and offers business applications in the rather large domain of enterprise software. Such applications may provide solutions for the most diverse problems around *Enterprise Resource Planning (ERP)*, such as *Product Life Cycle Management (PLM)*, *Supply Chain Management (SCM)*, *Supplier Relationship Management (SRM)* or *Customer Relationship Management (CRM)*. For the sake of clarity, our exemplary company focuses on essential business purposes in the context of CRM, which include the management of customer quotations, purchase requests and orders, product items in warehouses and the processing of payment tasks. Figure 3.1 illustrates common elements of corresponding sales processes and the spectrum of functionality that the software company covers. The figure's left side depicts consumers that either are contacted by a sales company or approach such a company by themselves, driven by desire. It shows how enterprises and their customers interact through sales quotations or direct purchase requests. Orders are executed by delivering corresponding goods to the customer and requesting aggreed payment. As presented on the figure's right side, the offered software application provides functionality to facilitate the management of business-related data, to execute necessary payment operations, the resolution of ordered goods to find a proper warehouse near to the address of a customer, and to integrate communication with customers via the internet or email.

Thus, the portfolio of our company covers almost all essential process steps that occur when a consumer wants to buy something from company that provides particular goods. Hence, the customers of the fictive company are enterprises that provide goods or services, and that require

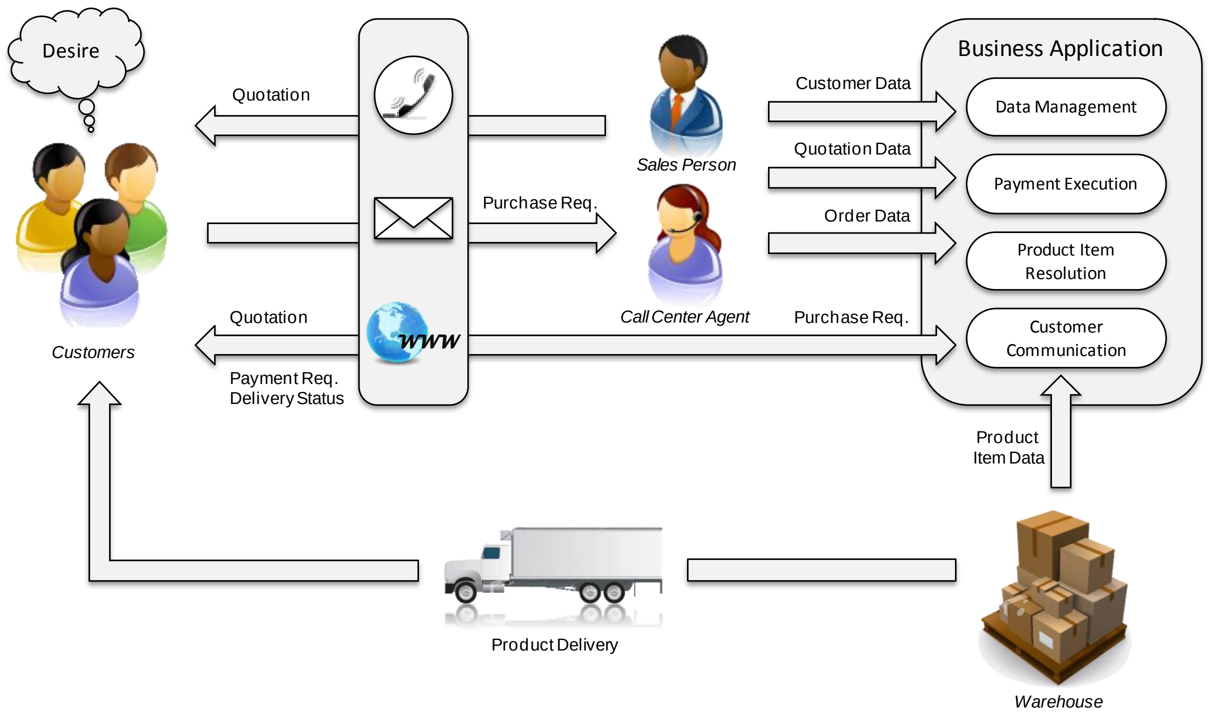


Figure 3.1: *Informal functionality overview of an exemplary business application*

software, which helps to manage business transactions as well as customer- and product-centric data arising from these transactions. Of course, each customer has individual requirements for the application that shall help to manage the offered business. Differences occur, for instance, with respect to the goods provided and their storage requirements, the handling of customer-centric information, the preferred way to accept customer orders, the range of accepted payment methods and the complexity of the network that performs business transactions.

To shed light on the challenges that arise in software engineering for our fictive company, Table 3.1 summarizes three possible customers with different businesses and requirements.

Customer A sells consumer electronics that are advertised with infomercials on television. The ordering process for this business is handled with external call centers that collect purchase requests and deliver corresponding information. Because Customer A acts as distributor for multiple companies, the offered choice of products changes frequently. Therefore, it would not be sensible to store customer-related data that goes beyond the information that is necessary for concrete orders, such as the customer's name and address. Additionally, Customer A maintains multiple warehouses distributed over the region of offerings. Hence, it is necessary to resolve the appropriate warehouse that is as near as possible to an ordering customer. To decrease the efforts for the distribution of ordered products and the costs for additional payment services, Customer A instructs an external parcel service to deliver desired goods and demand corresponding payment.

Customer B maintains an online book store in the internet. The entire web frontend that includes the receiving of customer orders is managed using an external web shop application. All books are stored in one single warehouse. As payment method Customer B provides the possibility to pay electronically by credit card. Therefore, it is crucial to store corresponding details about customers and their bank accounts, in order to notify an appropriate external payment service of the authorized transactions. Because Customer B uses the internet as distribution platform powered by an external web shop and maintains only one single stock, a supporting business application would not need to provide multi-user access as needed for Customer A.

Customer C is a single person that provides relaxing massages and offers home visits for 20 minute sessions in companies and for private individuals. Because the offered product is a service, no particular product items must be managed. There are also no special ordering or payment processes. Interested customers advertised by the homepage of Customer C or by word-of-mouth recommendations simply place a telephone call, arrange an appointment and pay cash after the massage. The most crucial functionality of a business application for Customer C are features to manage customer data. It is essential to store personal data, including the date of birth for congratulations, and business data, such as the number of massages that were performed. This way, Customer C is able to form customer groups and rate them accordingly to control the preference of particular customers.

Business Characteristics and Requirements	Customer A	Customer B	Customer C
Business	television sales of consumer electronics	online book store	relaxing massage for companies and private individuals
Product Items	in multiple stocks	in one single stock	human labor, no particular items
Accounting	customer addresses are bound to orders	storage of customer data necessary for payment	most important, including customer grouping and rating
Ordering	via external call centers	via the internet through an external web shop	via email and phone, appointments managed with external calendar
Payment Processing	cash on delivery via external parcel service	by credit card	cash after a massage session
Distribution	multiple call centers need system access to enter new orders	none, client server architecture managed by web shop	none, single end user desktop application necessary

Table 3.1: *Different businesses and requirements for a business management application*

The sketched scenario comprises a multitude of diverse challenges. Although our fictive company specializes only on a very small area of the domain of enterprise software, it becomes clear that an application which realises the requirements of Customer A, B and C all together must be extraordinary complex. All customers have entirely different technical constraints for such an application. While Customer A demands a system that must allow distributed multi-user access from different call centers, Customer B requests a server side application that contains appropriate interfaces for a corresponding web shop. All this would be by far too complex for Customer C, who should be completely satisfied with a standalone desktop application. Also the desired functionality differs massively between the three parties. The kind of business operated by Customer A and B is entirely different from the business of Customer C. This is already just because they sale countable product items instead of human labor. Nevertheless, the software required by Customer C needs to provide sophisticated account management features, which fits perfectly well into the product portfolio of the fictive software company.

The implementation of three different products on diverse platforms for all three customers is not a choice. The products would comprise a massive overlap in functionality and it would be too expensive to implement, test and maintain them in isolation. Hence, it becomes bold that a mechanism is necessary, which allows specifying the entire functionality portfolio of our fictive company (as illustrated in Figure 3.1) in one technology independent manner. Additionally, because of varying functional requirements of different customers, the company's portfolio must be easily divisible into sensible components, given the customers' requests. It is thus necessary, to perfom a sophisticated modularization, which is to be applied on a technology independent level. Furthermore, a transformation into executable software must of course also take place.

3.2 Efficiency in Software Production

In the preceding section we introduced a short example that should help to illustrate the tension in today's software companies. On the one hand, their is a multitude of alternatives for the technical realization of software. On the other hand, there are manifold customer requirements. In summary, we state that software industry in the twenty first century is confronted with enormous challenges. The rapidly proceeding technological advance in computer hardware recently opened and steadily opens new business options. In parallel, potential customers are only willing to pay for actually necessary functionality. This tension is aggravated by competitive market constraints, where inexpensive, fast and effective development is crucial for the survival of an enterprise.

In this section, we elaborate on several important problems in current software development. We claim that these problems cause a rise of the overall cost for production because of losses on both, the disbursement and the earning side of business. This renders today's software development inefficient and is one reason why software product revenues are lower than they could be.

3.2.1 Complexity and Knowledge Representation

One major reason for modest revenues in software companies are the overall costs for the development of these products. There are several reasons for high financial efforts in software development. In the following we summarize two of them, namely the complexity of development landscapes and the insufficient representation of development knowledge.

3.2.1.1 Complex Software Development

In recent years, market requirements for software products steadily grew as fast as the technological possibilities to implement these requirements. Constantly growing network bandwidth and storage capacity, new communication standards, such as *Radio Frequency Identification (RFID)* [Fin06] or Bluetooth [Blu08], innovative mobile devices [AAH05] and ubiquitous computing [Gre06] force software manufacturers to develop applications that integrate into extremely heterogeneous platforms. This requires huge technology portfolios to be available to solve particular customer problems. Developers have the choice between programming languages, function libraries, application frameworks, persistence technologies, communication platforms, development environments and design approaches. The diversity of solution alternatives is traversed by various application domains, such as web or desktop applications, embedded development, ubiquitous and mobile computing.

To conclude, software development today is a highly complex business, where participating people need to be aware of most diverse technologies. Coincidently developers should master at least one of these technologies to deliver particular solutions to an entire ensemble. The knowledge and intellectual power in different domains must be integrated technically and socially. This leads to tremendous financial efforts for IT companies.

3.2.1.2 Deficient Knowledge Representation

Besides technological diversity in the solution landscape, IT companies have to face further crucial challenges, such as the question how to capture individual development knowledge. All along in IT industry, it has been a common problem when key personnel leaves the company without having properly documented produced assets, designed architectures or, in short, the idea behind particular work. In such cases, valuable knowledge is lost with the farewell of employees. In consequence, new staff members need to dive deeply into existing structures, just to understand already hard-earned application code. To alleviate these problems, many companies establish guidelines and templates to document their solution landscape properly. Usually, such documentation patterns are sufficiently general to be applicable for the entire set of a company's assets. Unfortunately, this generality is often not the appropriate way to capture information that is specific to particular solution domains. For instance, a programmer might understand a behavioral pattern much faster with an illustrative state chart or activity diagram than by reading a written story about it. Several companies use more advanced representations for particular development assets that might even provide tooling for visualization, such as the Unified Modeling Language [UML07]. However, these representations rarely act as first class constructs in a system architecture and are often impoverished to outdated documentation material.

To conclude, the representation of development knowledge in IT companies goes too seldom beyond actual implemented code. In the case development assets are documented, this documentation is often too general to be effectively understood by developers that might be familiar with only a specific domain. This slows down the development process and already available technical expertise can hardly be reused, which leads to extra financial effort.

3.2.2 Variety of Customer Needs

The mismatch between customer needs and product features is another major reason for modest profits for software companies. This issue is again caused by deficiencies in the manufacturing of software but is, unlike the problems discussed in Section 3.2.1, mostly reflected on the earning side of business. Many companies provide products that do not exactly fit the customers' needs. The customer on the other side, does have individual requirements and is not willing to pay for insufficient or unnecessary features. This mismatch between customer requirements and product capabilities leads to a loss of potential customers. In the following, we summarize two reasons for this.

3.2.2.1 Restricted Requirement Formalization

The communication between product management and development teams is often slow, imprecise and ambiguous — also for small size enterprises. Even if an employee manages to record a customer's requirements as precise as possible, this is no guarantee that successive development teams implement these requirements in the intended way. To ensure an unbroken flow of information, companies employ variably detailed requirement documents, reaching from high-level *market requirements documents (MRDs)* and *product requirements document (PRDs)* to fine-grained *software requirements specifications (SRSs)* and even finer grained *software design descriptions (SDD)*. Such layered documentation architectures are error-prone and costly because the collected material only serves as documentation for the actual realization. Once noted information cannot *directly* be reused during software system specification. Instead, various participants in the development chain are forced to read, interpret and use this material as input for successive implementation steps. Each participant has an own technological and social background and interprets terms and interrelations individually even if there are constraining guidelines on how to perform this task (cf. Section 3.2.1.2).

To conclude, product innovation processes comprise tedious procedures from gathering customer requirements to actual product implementations. Yet, these procedures are massively backed with documentation material which is not useable as primary input for software development. This raises the probability for misunderstanding or misinterpretation and may lead to inaccurate requirement implementation and delays in production. As a result, customers are not satisfied and search for alternatives to the desired product.

3.2.2.2 Restricted Product Customization

Software product customers know their problems and want them to be solved. The usual buyer is price-conscious and does not want to spend more money than neccessary for an application that shall solve a particular problem. In this regard, companies are competing for the product that comes with the feature set that is at most suitable for a dedicated target group. In doubt, a cheap application that comes with only few features but provides indispensable capabilities is preferred to another product that also provides the desired capabilities but is expensive because of extra, unneccessary features. Here, one essential problem arises, namely the restricted capability of companies to concentrate on individual customer needs. Due to the complexity in software development (cf. Section 3.2.1) today's IT companies are rarely able to adjust their products to individual requirements. Product managers are forced to cluster customer needs into diverse target groups and align the software production accordingly. A particular customer can not at all or only poorly, decide which functionality the desired product contains. In case a sufficiently important customer has specific requests, a dedicated development team is established to implement a new solution. This in turn is because of its individuality often not or only partially adequate for other target groups in the remaining product portfolio.

To conclude, potential customers are not able to influence the feature set a product provides. In parallel, individual solutions for particular customers are not enough reusable for the remaining product portfolio. Consequently, prospective buyers are not satisfiable with provided product features and are lost as customers for the company.

3.2.3 Summary

In order to summarize the problems that were identified in this section, Figure 3.2 illustrates a problem hierarchy that was created following the guidelines of a prominent problem analysis approach of the *German Technical Cooperation (GTZ)* called *Zielorientierte Projektplanung (ZOPP)* [HG97] or *Objectives-oriented Project Planning (OOPP)* in english. The hierarchy is to be read along arrow direction as “leads to” or against arrow direction as “is caused by”. The hierarchy reflects the introduced problems in the context of the given economic consideration. To this end, typical engineering challenges lead step by step to the concluding main problem of modest net profits for software products in IT companies.

The generally high level of heterogeneity in development landscapes of software companies causes enormous costs because of significant development and maintenance efforts. This complexity is not managed on a sufficiently abstract level and often leaves no place for a proper separation of domain-specific concerns. This entails that new employees need too long to become aquainted with available implementations and to solve particular problems also if implementations are already well known. The result is a slowed down development process that causes extra costs. After having rolled out a product, customers are often discouraged because they do not find the desired functionality. This is either because desired product features are missing

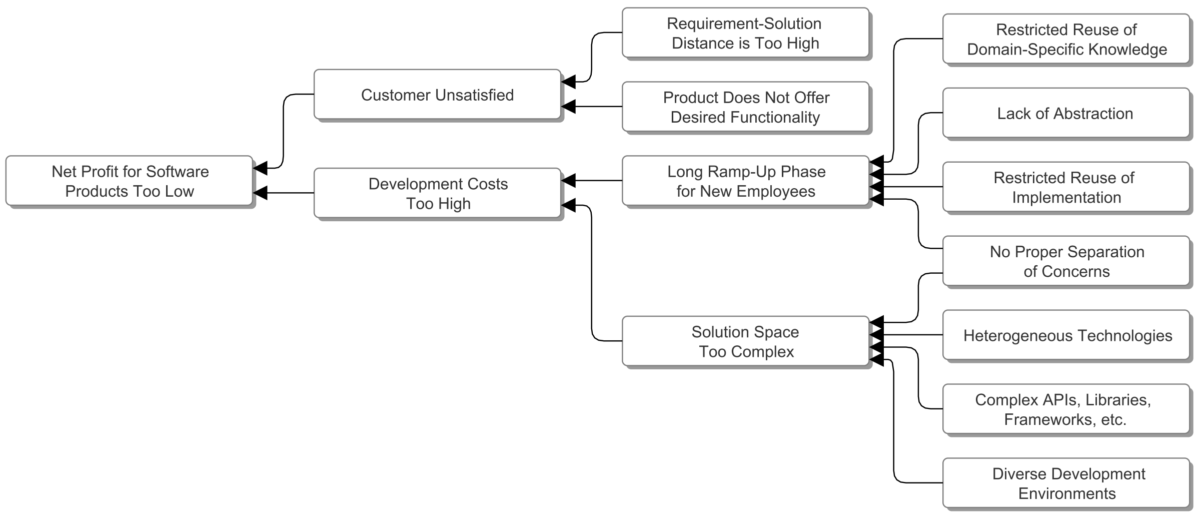

Requirement-Solution Distance is Too High
Product Does Not Offer Desired Functionality
Customer Unsatisfied
Net Profit for Software Products Too Low
Development Costs Too High
Long Ramp-Up Phase for New Employees
Solution Space Too Complex
Restricted Reuse of Domain-Specific Knowledge
Lack of Abstraction
Restricted Reuse of Implementation
No Proper Separation of Concerns
Heterogeneous Technologies
Complex APIs, Libraries, Frameworks, etc.
Diverse Development Environments

Figure 3.2: *Summarized problem hierarchy*

or simply too far away from the customer's expectations. While the former prevents potential customers from buying a product, the latter also causes additional costs for product support.

3.3 Vision Outline

In the Sections 3.1 and 3.2 we elaborated on the actual challenges for today's software companies concerning problems in engineering environments, processes and tools. This section provides an outline for our vision to alleviate the identified problems. To this end, we present general aims for this vision and coin the notion of *Multi-Domain Engineering (MDE)*.

3.3.1 Goals and Technologies

To design an approach that overcomes the problems presented in the two preceding sections, we started by developing a goal hierarchy that was directly derived from the problem hierarchy in Section 3.2.3 (see Figure 3.2). Figure 3.3 shows the resulting graph. Almost every problem of the problem hierarchy relates to a particular goal in the depicted graph and will thus be solved by our approach. The arrows in the diagram are to be read as “leads to” along arrow direction and as “is solved by” against arrow direction. To emphasize that sub goals are preliminaries to the aggregated parent goals, we changed the graph orientation from left to right compared to the problem hierarchy.

As illustrated, it is necessary to increase the value of documentary assets and requirements in parallel with a reduction of the importance of actual programming code. This way, the solution space becomes less complex because the level of abstraction in development is raised. A proper separation of concerns and an enhanced binding of dedicated requirements to solution space artifacts clear the way to domain-specific development. Altogether this entails a reduction of costs for development. To also satisfy individual customer needs, the explicit specification of these needs must be possible and modifications to a product's architecture or implementation in general must be flexibly realizable.

Following to the designed goal hierarchy we identified enabling technologies that allow for its implementation. To this end, we selected three emerging paradigms in software development research and practice, namely *Model-Driven Software Development*, *Domain-Specific Modeling*

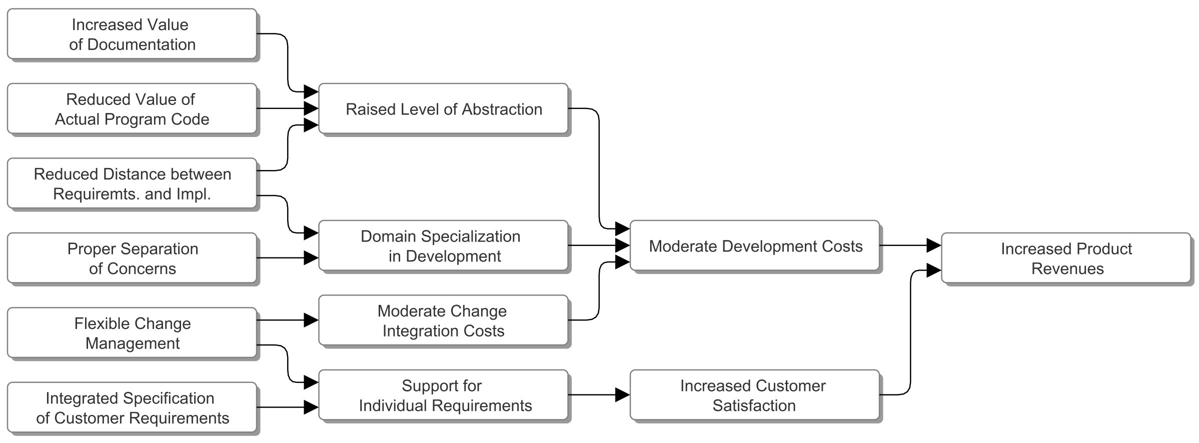

Increased Value of Documentation
Reduced Value of Actual Program Code
Reduced Distance between Requiremts. and Impl.
Proper Separation of Concerns
Flexible Change Management
Integrated Specification of Customer Requirements
Raised Level of Abstraction
Domain Specialization in Development
Moderate Change Integration Costs
Support for Individual Requirements
Moderate Development Costs
Increased Customer Satisfaction
Increased Product Revenues

Figure 3.3: *Summarized goal hierarchy*

and *Software Product Line Engineering*. In the following, we summarize the advantages of each paradigm to underline its appropriateness for the implementation of our goal hierarchy. A comprehensive introduction to all of them is given in Chapter 2.

Model-Driven Software Development improves the ability to integrate extraordinary heterogeneous technologies through the abstraction from platform-specific programming code to coarse-grained models. To this end, diverse platform-specific generators that are once written, tested and optimized, can be steadily reused for different projects. Each generator can be employed with software specification models that only need to be specified once. This saves time and fosters reuse of modeling facilities.

Domain-Specific Modeling provides languages that are tailored to particular application areas and problem domains. By employing such languages, developers' efficiency can be raised, while production processes get more error resistant. Moreover, through the employment of multiple domain-specific models, a complete software specification becomes much more manageable. Instead of maintaining one huge software model written in, e.g., UML, different problems can be solved step-by-step with dedicated languages. Each problem area can be delegated to corresponding stakeholders with appropriate domain knowledge. This ensures a proper separation of concerns in development.

Software Product Line Engineering treats the notion of variability as central development aspect and facilitates the alignment of software products directly to customer requirements. The development of applications through the composition of core assets is predominantly advantageous for the reuse of feature-centric implementations. Because the assets are directly bound to product features with a clear business intention, they remain handy and reusable for diverse product solutions. This improves both, product customization capabilities and the pace of product development for already implemented application domains.

3.3.2 Multi-Domain Engineering

In the previous section, we rendered three emerging software development paradigms as ideal candidates to implement the goal hierarchy depicted in Figure 3.3. While MDSD increases error-safety and platform-independence, DSM especially with multiple DSLs fosters separation of concerns and efficient, problem-centric implementation. The focus on variability in SPLE improves flexibility and facilitates fast and effective development of customer-centric products. Yet, the different paradigms exist in isolation and need to be integrated, as already revealed in several research activities [CA05, HL06, Fea07, Con07a, VG07, HKW08]. These efforts predominantly aim at the integration of modeling and product line technologies by developing approaches to combine feature models with system specification models. Within this thesis we step in following to this binding process and coin the notion of *Multi-Domain Engineering (MDE)* as follows:

Multi-Domain Engineering strongly emphasizes the advantages of model-driven software development with *multiple* domain-specific models as a neccessity for efficiency in software development. Furthermore, it treats the paradigm of software product line engineering as indispensable means to achieve a maximum degree of reuse and flexibility. Consequently, it denotes model-driven development with multiple domain-specific languages in variabilityintensive scenarios.

In this thesis, we focus on *domain engineering* which is the development step that delivers the foundation for product lines in particular application domains (cf. Section 2.3.3.1). Here, we are aiming at a seamless integration of domain-specific units without harming the limits of each solution domain. We want to establish and maintain an explicit representation of the knowledge about the semantics of connected units in order to propose proper connection interfaces between them. The connecting network must be smart enough to react on modifications in domain-specific models and ideally adapt all system parts that are affected by these modifications automatically. To finally arrive at executable software programs, the network must also be usable to derive integration possibilities on code level.

Figure 3.4 illustrates our vision based on a product line development scenario. According to the SPLE approach the figure is horizontally divided into problem and solution space. In vertical direction different layers known from the MDSD paradigm are depicted. In this respect, we have divided layer M1 into two distinct levels that contain the *Variant Independent Model (VIM)* and the *Variant Specific Model (VSM)* [HL06], respectively. The figure's left side shows (from top to bottom) how (1) the requirement analysis yields (2) a product line feature model, (3) a variant model that is a subset of this model for a dedicated product, and (4) the concrete product that results from transformation processes in the solution space. The figure's right side illustrates the product line's solution space containing (from top to bottom) (1) multiple DSLs that are semantically enriched, (2) the modeling landscape comprised of various DSMs that are properly interconnected, (3) a subset of this landscape according to the selected variant model, and (4) the finally generated executable software.

Following this approach, i.e., facilitating an integrated multi-language scenario, allows to unfold the entire impact of the advantages of MDSD and DSM. Each field of technical expertise can be enhanced with tools that enable developers to specify individual knowledge with a maximum degree of expressiveness. Model-driven concepts facilitate platform-independence. Although we cluster our system into domain-specific solutions, our connecting semantic network provides necessary capabilities to focus on variability-driven engineering by actively solving inconsistencies between connected units.

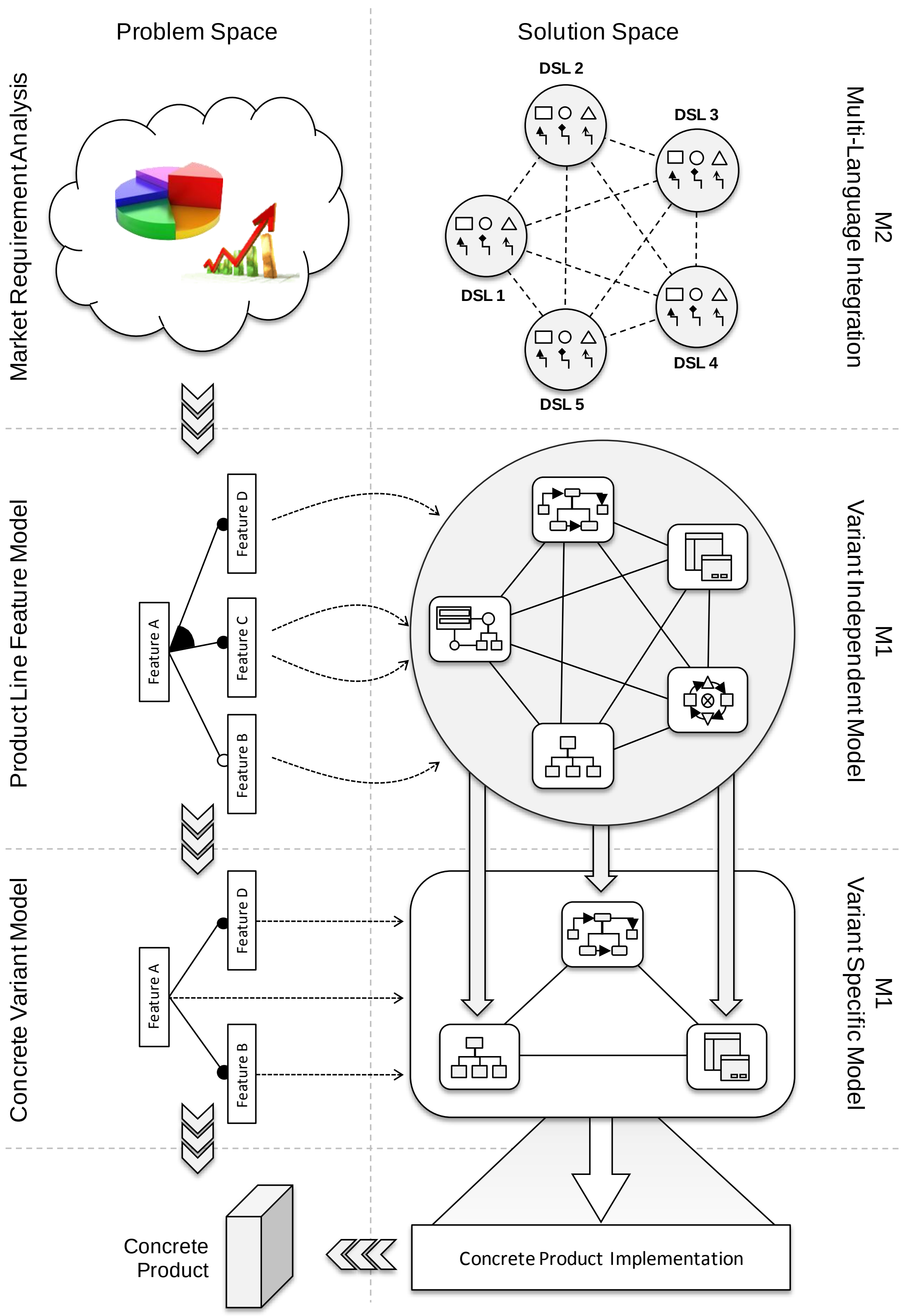


Figure 3.4: *Multi-domain engineering vision overview*

3.4 Essential Requirements

The previous sections discussed problems in today's software development and goals for an approach that shall help for their alleviation. After having introduced our vision outline in Section 3.3.2, this section reveals concrete requirements to achieve the goals to achieve in order to implement this vision (see Section 3.3.1). To this end, recall the motivational example from Section 3.1. Regardless of which runtime platform realises one of the desired customer products, the functionality of the product portfolio would be comprised of several distinct solution parts, such as data structure descriptions, behavior specifications, persistence technologies or user interfaces. Each part interacts with and thus depends on one or more other parts. A user interface, for instance, might indicate the actual state of a customer order in the exemplary business application. Such entities in turn, might only be participants in a globally orchestrated business process that controls additional tasks, such as product item resolution or customer communication. The actual business logic again, defines how both, internal states of data entities and process flows, shall react on user input or system events.

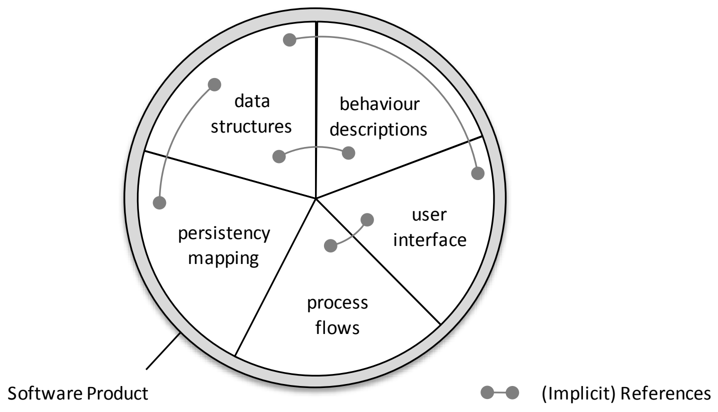


Figure 3.5: *Different solution domains in complex software products*

In Figure 3.5, we illustrated a sufficiently complex software product with accordingly meshed solution parts. As depicted, each solution part maintains references to other parts. These references can be of entirely different nature depending on employed programming languages and software platforms. For example, in web application development, as with *J2EE Enterprise Java Beans (EJB) [Mic08]*, it is common to apply string-based references between XML configuration files and business logic code implemented in Java. Instead, in software applications that build on only one single programming platform, such references might exist implicitly, e.g., through type imports or method calls resolved by the compiler for the employed statically typed language.

In the following sections, we analyze this abstract solution space description of the exemplary product portfolio to step by step derive requirements that are to be fulfilled, if such a solution space shall be implemented following our vision of multi-domain engineering. Recall from 3.3.2, that each of the depicted solution parts forms a particular area of expertise that shall be expressed by means of an appropriate domain-specific language. Multiple domain-specific models written in these abstract languages shall than finally form the entire solution space. Corresponding platform-specific generators facilitate the production of executable runtime code for each solution part.

3.4.1 Language Design

The first important step in MDE development is the creation of appropriate domain-specific languages. To this end, it is necessary to (1) identify fitting solution parts, and (2) specify an appropriate DSL for each solution part including a metamodel, constraints and modeling tools (see Figure 3.6). The identification process must lead to significant solution domains that are both, sufficiently complex to justify the design of dedicated languages, and limited enough to serve as adequate candidates for domain specificity. After having achieved to identify suitable language domains, appropriate requirements must be specified that a language in each domain must meet for a given context. Next, the process of creating domain-specific languages starts. For each solution domain this process will result in an appropriate metamodel, additional constraints and modeling tools. In this respect, it has to be ensured that the resulting metamodel is change-safe to a sufficient degree. If it is not, all assets that build on this metamodel have to be revised whenever new models discover unforseen deficiencies.

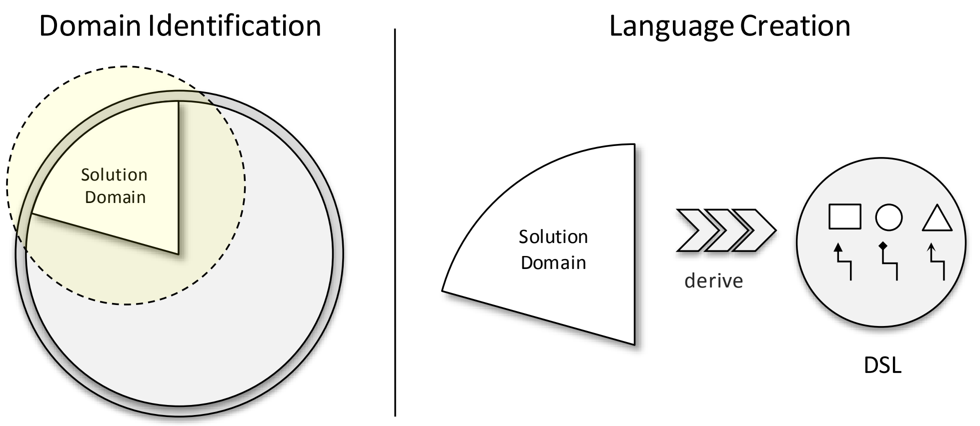


Figure 3.6: *Domain identification and language creation*

Especially for developers who do not have experience as language designers, these are substantially challenging tasks. Hence, a comprehensive approach to multi-domain engineering must support the language creation task to ensure that developed DSLs are to a certain degree complete, sufficient and change-safe. This leads to the first requirement for MDE.

Requirement 1 *MDE needs a dedicated development process that supports the design of domain-specific languages. This process needs to assist in both, identification of appropriate solution domains and actual language creation.*

3.4.2 Domain Interaction

Once appropriate solution domains have been identified and the corresponding languages have been created, the most important step towards model-driven development with multiple DSLs is accomplished. Unfortunately, the resulting abstraction from actual code level implementation entails additional problems. An obvious observation is that the lifting of previously implemented code structures to more abstract DSMs, does not liquidate implicitly or explicitly contained references and dependencies between solution parts (cf. Figure 3.5). In fact, because of abstraction these dependencies grow to complex relationships between languages that are instantiated by concrete references between models. Figure 3.7 illustrates this problem in multi-DSL development for generative scenarios.

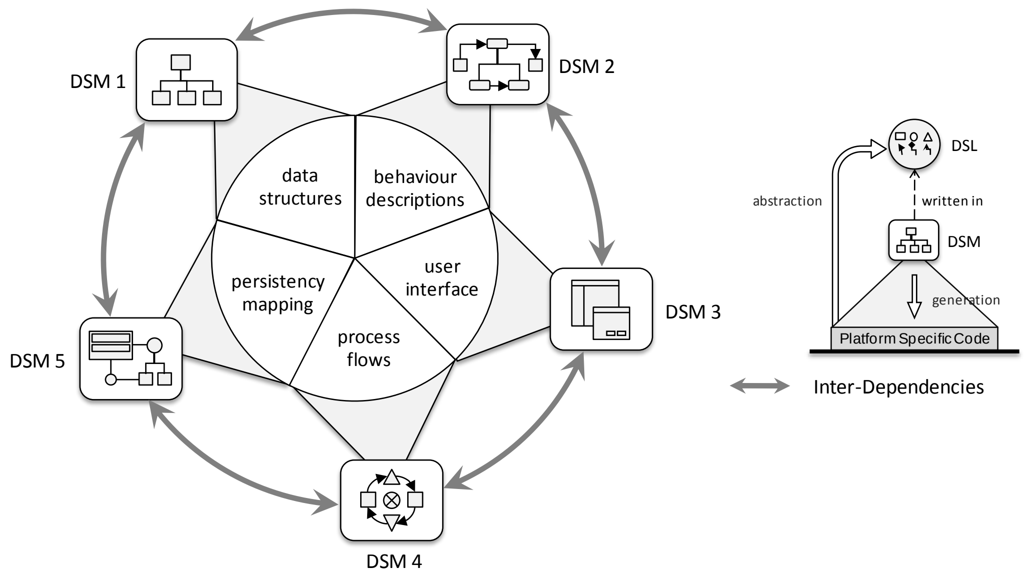


Figure 3.7: *Dependencies between DSMs in multi-domain engineering*

As introduced in Section 2.2.3, there are two ways to resolve inter-language dependencies. In interpretive systems it is up to the underlying platform to identify and process actual references between DSMs. In generative systems, this task is performed manually by transformation developers who implement model-to-model and model-to-text transformations and know about the interaction of adjacent entities. Either way, there are no means to ensure referential integrity and detect inconsistencies between models in the solution landscape. Interpreters do not ensure that inter-model references are consistent and valid. Instead, invalidity leads to false system behavior, which is revealed only at runtime. If transformation developers implement inter-model references, inconsitent modeling assets end up in erroneous generated code or incomplete target models. In addition, dependencies between languages are described implicitly in transformation templates and remain hidden for further use.

However, the discovery of false references and inconsistencies in the solution landscape, as early as when creating the modeling facilities, is crucial for efficient development. This is especially true in the product line context of MDE, where feature-driven modifications in existing modeling assets frequently harm integrity. To achieve this early discovery, inter-language dependencies and inter-model references, that are usually implicitly hidden in interpreters or transformation templates, must be formalized and made available to corresponding consistency processing. This formalization needs to allow not only pairwise references but must rather facilitate consistency processing between multiple modeling artifacts in parallel.

Requirement 2 *MDE needs an explicit representation of both inter-language dependencies and inter-model references. This representation must be amenable for techniques that allow the discovery of false references, inconsistencies, and hence can ensure referential integrity.*

Requirement 3 *The explicit representation and corresponding consistency processing needs to facilitate the ensurance of referential integrity between a multitude of different models and must not be limited to pairwise calculations.*

3.4.3 Integration of Execution Semantics

Once the required languages are designed and the necessary support to consistently connect corresponding models in the solution space is available, the software specification needs to be delivered. Domain-specific developers can unfold their expertise and contribute different parts to an entire story. Yet, a complete and consistent model specification based on multiple DSLs is not sufficient to finally end up in running software applications. At this point in the development process it is not clear how modeled information can finally be leveraged.

In interpretive multi-DSL scenarios, development would end at this point, as such scenarios are based on an interpretive platform that is aware of all participating languages and, hence, able to process modeled data. Unfortunately, interpretive scenarios are rather an exception than the rule. While there are prominent examples in web application development (see Section 2.2.3.1), for instance, conventional GPL-based software at best employs particular interpreters for *selected* concerns only. Examples are recently emerging declarative user interface languages, such as the *eXtensible Application Markup Language (XAML)* [Nay07], or behavioral frameworks, like *Commons SCXML* [Fou09c], an implementation of the *State Chart XML (SCXML)* working draft published by the *World Wide Web Consortium (W3C)*.

Also in generative scenarios appropriate templates could produce programming code that integrates into such a framework, even though this is not obligatory. No matter whether interpreter-based or simple GPL-based generated code is employed, diverse implementation parts must be integrated properly. This is because the resulting software application must implement an unbroken control and data flow, that connect various domain-specific code parts (see Figure 3.8).

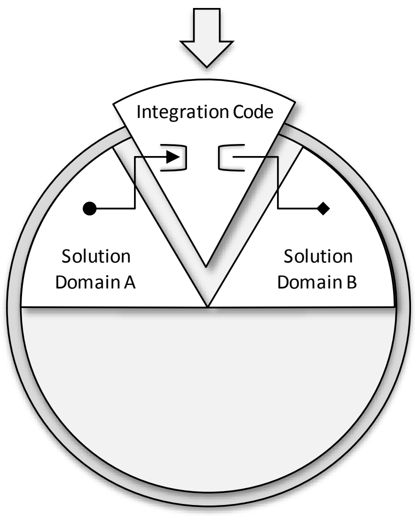


Figure 3.8: *Providing integrating code structures*

The integration task is limited to those places in the modeling landscape, where different models reference each other. Each composition part on code level maps to a corresponding reference on model level. For every type of reference a certain type of composition part could be required. This is why developers must be able to provide integrating code structures for particular reference types between employed languages. Of course, compositional code structures depend on the underlying platform for each implementation that is to be integrated. Additionally, it is likely that, depending on the kind of employed languages, particular code structures follow a certain pattern or can even be reused without modification. Therefore, a mechanism that proposes selected integration structures based on the platforms to integrate and the type of language reference would be very helpful.

Requirement 4 *An MDE approach needs an integration framework that facilitates the definition of proper code composition structures for particular language dependencies. Here, the framework should provide additional help to select appropriate composition patterns depending on kind and complexity of particular connections as well as the underlying runtime platforms.*

3.4.4 Language Administration

Introducing modeling technologies in companies is, of course, not for free. The implementation of domain-specific languages, corresponding modeling tools and code generators or inerpreters causes additional costs. Although the extra efforts pay off quickly [Bet02, Met07], it is crucial to ensure that once developed modeling facilities can steadily be reused in the most diverse contexts. For instance, if the fictive company from the motivational example of Section 3.1 would decide to create new products that are not related to the domain of business applications, it would be desirable to use the DSLs for software specification that are already available instead of reinventing the wheel for the new project. Naturally, this does only work for DSLs that are applicable to the new product domain, which would apply at least for DSLs that are of technical nature. A DSL to describe data structures, for example, would be applicable in both, the domain of business applications and the domain of computer games. Because different projects have different requirements it might be that existing DSLs needs to be equiped with for example additional code generators or new model editors. It is also possible that a required DSL would not be available and would have to be developed from scratch. Such additional assets must not be lost, once they are implemented (see Figure 3.9).

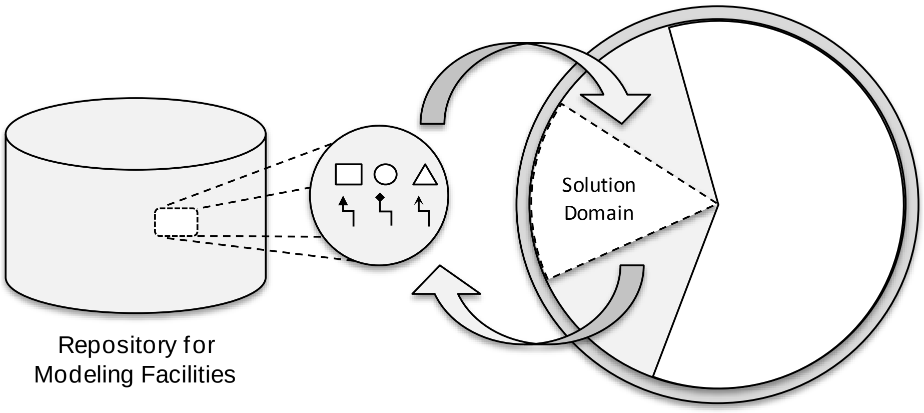


Figure 3.9: *MDE repository for modeling facilities*

To this end, it is necessary to establish a common way to store existing modeling facilities in a central repository. This repository must be open for browsing in order to find appropriate languages for particular purposes. Additionally, it must be possible to complete modeling facilities of existing domains with additional assets.

Requirement 5 *In MDE, modeling facilities, such as domain-specific languages, modeling tools, code generators or documentation, needs to be organized in a central repository. This allows efficient reuse and raises the value of these facilities.*

3.4.5 Implicit Language Semantics

Currently available modeling technologies (cf. Section 2.1) facilitate the development of modeling languages. To this end, they offer appropriate meta-metamodels that provide core entities for language creation, such as classifiers, attributes and references. However, corresponding frameworks do not provide support to derive the meaning or purpose of a created language. The sole assertions that can be made about a language's entities is derivable from their metatypes. For instance, a modeling language for user interfaces might contain an entity that represents textfield widgets. Based on the information of the language's meta-metamodel, it can only be deduced that this textfield entity is a concept, i.e., a classifier that is characterised by further attributes. That a textfield in a software application usually shows information about actual data entities and represents an input channel for the end user, is not derivable. As with language classifiers, there is no way to derive additional information about the meaning and purpose of attributes and references in a modeling language.

Hence, no assertions about single entities and relations of a modeling language can be made from the perspective of the application context in which the language shall be employed. This means that the actual role of a language in such a context remains blurred until a developer looks at it and interprets the language based on his/her own experiences. This leads to several problems in development with an integrated MDE approach and in MDSD in general. First, when working with a language repository (cf. Section 3.4.4) it is hard to find appropriate languages for particular purposes. For example, a project manager could not query such a repository for, e.g., languages that are useful for the design of system behavior. Instead, it would be necessary to browse the entire repository or perform an inaccurate syntax-based search. Second, in a multi-DSL scenario a modeling language shall be integrated with various other languages. A lack of information about how the language arranges with other languages renders it difficult to define proper interaction and dependency interfaces, as demanded by Requirement 2 (see Section 3.4.2). Third, language integration on the implementation level might be dependent on the type of references between DSMs and DSLs on the modeling level. Without proper attribution of modeling entities it is impossible to significantly classify code integration patterns, as it is demanded by Requirement 4 (see Section 3.4.3). Figure 3.10 illustrates the equipment of DSLs with semantic information.

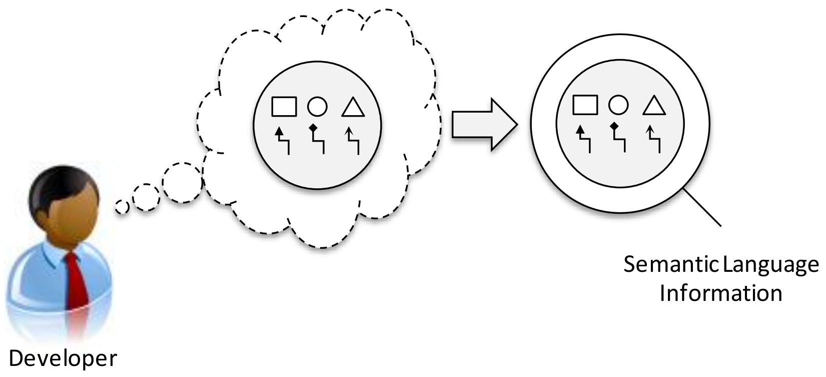


Figure 3.10: *Equipping DSLs with semantic information*

Additional information about the semantics of modeling languages is important to quickly get an explanation of their role, purpose and qualities. It is furthermore crucial to provide significant proposals for possible inter-model references and appropriate code integration patterns. To this end, we need to provide a framework to record the *meaning* of modeling languages in the context of software applications.

Requirement 6 *An approach to multi-domain engineering must provide appropriate means to classify constructs of modeling languages to determine their role, purpose and qualities in the context of software applications.*

3.4.6 Language Maintenance

An MDE approach that meets all above listed requirements already provides support to create languages, to integrate these languages, to interconnect models based on them, to derive executable programs from this input and to reuse modeling facilities in various contexts. We will now focus on reuse in model-driven scenarios. This is crucial to raise the value of modeling facilities and to ensure that model-driven techniques are indeed applied. In this context, it is of vital importance to consider maintenance aspects that emerge during the evolution of solution domains. In case a DSL's metamodel does not fulfill necessary requirements and therefore must evolve, all referring assets that the DSL is integrated with, would be affected as well. This involves interaction and dependency interfaces to other languages, existing references from DSMs of the evolved DSL to DSMs of other languages and integrating assets that combine solution domains on the implementation level.

In Requirements 2 and 6, we justified the need for an explicit, semantically rich representation of a DSL's ingredients as well as their interdependencies to other DSLs. From the maintenance point of view introduced in this section, the *scope* of such a representation has to be considered. In particular, it has to be questioned whether DSLs must be represented in their entirety with a corresponding formalization. Given that this would be the case, each and every language modification would affect this formalization which would also harm the integrity of dependencies and references to other languages. This would entail significant efforts when maintaining the corresponding language representations and references. Even before taking the modification of DSLs into account, a repository-centric architecture (cf. Section 3.4.4) allows to search for language alternatives. If a DSL does not fit the requirements of a given project, the first action *before* DSL modification must be a comprehensive repository search.

Hence, the in the Sections 3.4.2 and 3.4.5 claimed explicit, semantically rich representation of DSLs and their relationships should *not* involve the entirety of all lanuage ingredients. Instead, it should be limited to information that is actually necessary to comprehensively describe language relationships and, finally, derive running software applications. Additionally, in case a DSL is insufficient for particular purposes, an MDE development process, such as claimed in Section 3.4.1, must foster an extensive use of existing modeling facilities. All this minimizes the impact of language modifications with respect to adjacent assets and, hence, improves the value of MDE.

Requirement 7 *A formalized representation of the meaning of DSLs in MDE must be limited to information that is necessary for language integration on M2-, M1- and implementation level. An MDE development process must, furthermore, incoporate the repository for modeling facilities as prior step to language modification.*

3.4.7 Guidance and Change Adaptation

Software *is* complex and so is the variety of modeling facilities in a multi-domain engineering environment. Although our MDE vision simplifies software development, improves the separation of concerns and lowers the distance between realization and the mental model of developers, software development *remains* complex. For instance, during the system modeling process, when different domain-specific modelers contribute partial implementations to an entire product, it is natural that inconsistencies between diverse models arise. Because of domain-specialization it is obvious that an expert for one domain must not necessarily understand insufficiencies that arise in other domains. Additionally, in a variability-driven context, as present in MDE, modifications are common also when the modeling landscape is not actively changed by any developer. So, the inclusion or exclusion of particular product features entails adaptations of modeling assets that belong to a feature's implementation. Both, active and passive influences to the modeling landscape cause inconsistencies. These, in turn, entail succeeding manual adaptations along dependency relations between participating models, which causes huge efforts during development.

Therefore, it is of vital importance that domain experts are supported in both, active and passive scenarios. When system modelers establish DSMs for a certain domain, they must be guided as good as possible. To this end, an MDE environment should propose modeling alternatives, which lead to valid and consistent models. In case of (e.g., variability-driven) passive modifications, corresponding mechanisms should facilitate the automatic resolution of inconsistencies to as large a degree as possible. If automatic resolution is not possible, manual interaction should be requested by the system.

Requirement 8 *System modelers that work with an MDE environment must be guided based on the semantics of corresponding modeling languages. Passive changes to the modeling landscape must be as far as possible automatically handled.*

Chapter 4

Solution Approach Overview

In Chapter 3, we provided a detailed problem analysis and outlined our vision of the *multi-domain engineering* paradigm that shall support the development of software and in particular software product lines. The paradigm emphasizes development with distinct DSLs in parallel, while continuously ensuring consistency in each domain and across domain borders. Through automatic adaptation of connected units in case of modifications, developers are massively supported in product line development.

In this chapter, we scope our solution to implement MDE and give an overview of the approach we follow, by introducing the major components of the overall architecture. We close the chapter with a summary of the benefits we expect of the approach.

4.1 Reviewing Requirements

In Section 3.3, we identified the integration of model-driven software development, domain-specific modeling, and software product line engineering as one way to solve the problems summarized in Section 3.2. We outlined essential requirements for systems that shall implement the envisaged MDE vision in Section 3.4. In summary, the most important requirement for a realizing architecture is the *establishing of a centralized binding network* to:

- capture the meaning of modeling constructs in domain-specific languages (cf. Section 3.4.5, Requirement 6)
- facilitate the description of relations and inter-dependencies between adjacent DSLs and to provide necessary means for the interconnection of models (cf. Section 3.4.2, Requirement 2)
- allow for advanced processing to ensure consistency and referential integrity between connected modeling assets (cf. Section 3.4.2, Requirement 3)
- allow for advanced processing to provide guidance and adaptation features in case of modifications to the modeling landscape (cf. Section 3.4.7, Requirement 8)
- enable for the integration of execution semantics of DSLs to derive executable systems from holistic modeling landscapes (cf. Section 3.4.3, Requirement 4)

Considering the listed requirements allows to estimate the necessary realization effort as well as the demands for enabling technologies. In the following, we implement this to further scope a platform that realizes our MDE approach.

4.1.1 Variety of Modeling Technologies

Now, one central aspect with a great effect to a realizing system needs to be clarified first, before analysing the listed requirements further. In previous sections when discussing MDSD we did not focus on dedicated modeling technologies or comprehensive frameworks that provide meta-metamodels and appropriate tooling to create modeling assets. This is simply because many of them exist, some in publicly available open source projects, like EMF with Ecore [BSM$^+$04] or the AMMA platform [INR09] with its *Kernel MetaMetaModel (KM3)* [JB06], some standardized or published and proprietarily implemented, like the MOF-based *Modeling Infrastructure (MOIN)* [Net07] from SAP [AG09] or GOPPRR used by metaCASE [Met09a], and others that stem entirely from commercial vendors, such as Microsoft's *MSchema* [You09]. All of them have their raison d'être with different strengths and limitations.

Hence, from the perspective of a comprehensive MDE vision, we can neither prefer nor omit particular technologies. This implicates that the demanded centralized binding network of MDE needs to be independent from any modeling technology, which has an effect of the realization of Requirement 6. To capture the meaning of DSL constructs, it is actually necessary to equip particular modeling assets with additional metadata that semantically qualifies these assets. Because we can not limit our MDE platform to dedicated modeling technologies, we can likewise not use appropriate metadata features that come with these technologies, such as EMF annotations for Ecore-based models [BSM$^+$04]. In other words, the necessary metadata must be kept separate from modeling technologies. Figure 4.1 illustrates DSLs of diverse modeling technologies with appropriate metadata held separately in the binding network.

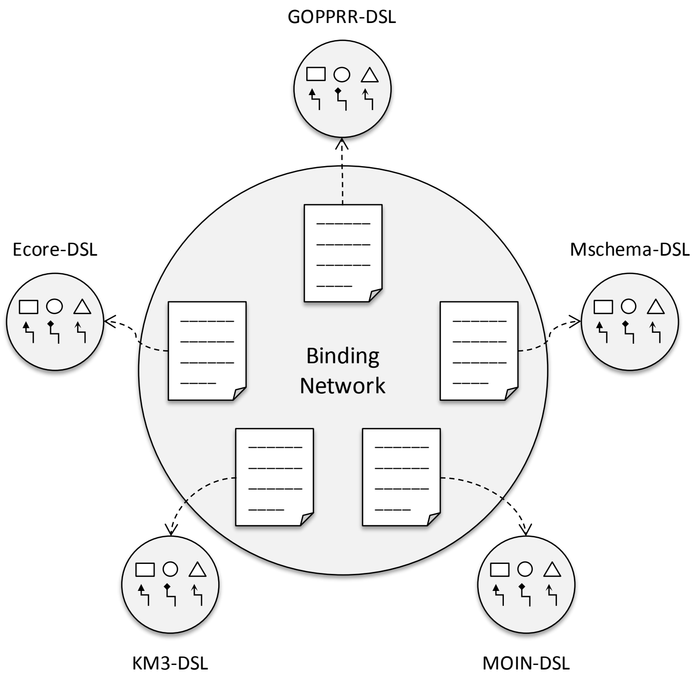


Figure 4.1: *Support for various modeling technologies through external metadata*

4.1.2 Loose Connections

Regardless of what technology is used for the binding network and whether it differs from integrated modeling technologies or not, another question arises from Requirement 2. It requests the establishment of references between DSLs and their instantiating models. There are several solutions for implementing this [LH09]. One possibility, is to use *explicit typed references*, i.e., employ the means of the underlying modeling framework. A second alternative is to make use of *string-based soft references*, i.e., references that are established based on string-equivalence in attribute values, as it is often used in interpretive scenarios known from web application development (see Section 2.2.3.1). Both alternatives are illustrated in Figure 4.2.

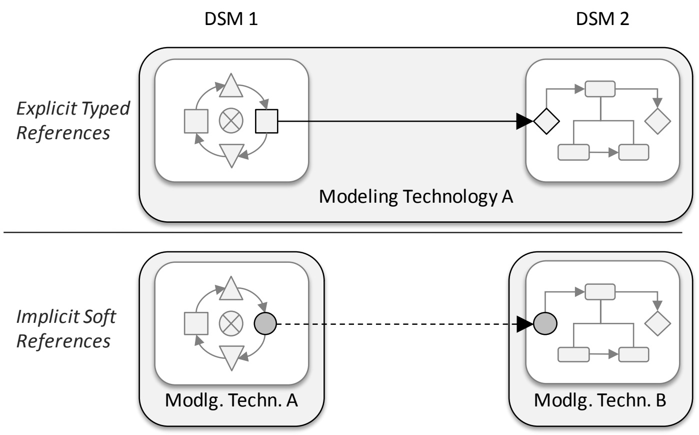


Figure 4.2: *Different reference types between models*

Now, the first approach obviously restricts the modeling landscape. If typed references between models are used, it is not possible to create a modeling landscape based on *different* modeling technologies, which limits reusability. Even though this might not be necessary or adequate in most use cases, the use of typed references still results in huge monolithic models that cause poor performance during editing or input/output operations [LH09]. When using string-based references instead, one major problem arises from the lack of omitted type information. As such references are only string values, according reference verification mechanisms would have to be developed from scratch. Additionally, such references severely affect the modeling assets. Languages need to declare appropriate identifier attributes and models would have to adhere to the employed reference scheme in the attribute values.

Both, explicit typed references and implicit string-based soft references, are difficult to integrate with an external binding network. In either way, we would need to analyze and extract reference information from the modeling landscape in order to rebuild it in our binding network, with different implementations for different modeling technologies. As a solution we propose a loose coupling approach by implementing *out-in references*, illustrated in Figure 4.3. To this end, the information about inter-model references is entirely stored in the central binding network. This in turn, maintains unique identifiers that point to dedicated modeling assets. This way, we (1) do not pollute the modeling landscape with reference information and (2) reduce the implementation effort to the extraction of unique identifiers for every employed modeling technology.

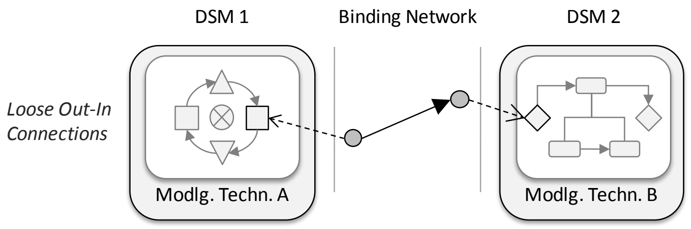


Figure 4.3: *Loose out-in references between models*

4.1.3 Consistency and Integrity

Although the requirements considered above were important for a comprehensive MDE architecture, proper solutions were rather easy to find and required merely profound engineering know-how than extensive theoretical foundations. In contrast, the Requirements 3 and 8 demand more sophisticated solutions. They embody the need for advanced reasoning capabilities in the MDE modeling landscape to (1) ensure that referential integrity and consistency are given and (2) provide additional support for developer guidance and automated model adaptation in case inconsistencies persist.

As the paradigm of model-driven software development is rather young, the need for *advanced* logics processing in MDSD arose only in recent years [WSNW06, HCW07]. The first remarkable specification that aimed at the integration of simple constraints is documented with the *Object Constraint Language (OCL)* [RP97] as part of UML, which is "a formal language used to describe expressions on UML models" [OCL06, p. 5]. OCL is one approach to alleviate the rather poor support for logics specification in MDSD. While it allows to specify constraints, abstractly implement algorithms, or to pose queries against models, knowledge deduction is not to be counted to the goals of OCL. Hence, OCL allows to ensure consistency of modeling assets but does not facilitate the derivation of necessary steps to get from inconsistent to consistent states. This renders OCL inadequate as an enabling technology for reasoning in MDE.

More sophisticated research concerning logics processing was done in the area of computational logics. Here, nearly 30 years of experience in research fields, such as *Description Logics (DL)*, led to comprehensive approaches with several international standards [KC04, BG04, PS08, SWM04], rich tool support [CP09, Jen09, SMI09, Rac09, TH09], and vibrant communities around the *Semantic Web* [W3C07]. Especially the strong theoretical foundation in research but also the comprehensive tool support renders Semantic Web technologies as an optimal alternative to implement the reasoning capabilities in our MDE architecture.

4.1.4 Execution Semantics

The challenge to integrate execution semantics of different DSLs strongly depends on the nature and characteristics of the established binding network. As demanded in Requirement 4, an MDE platform must provide means to derive composition structures that integrate different solution domains on code level. To this end, a comprehensive analysis of the explicit information about the referential interplay available from the central binding network is necessary. As concluded in the previous section, Semantic Web technologies provide a thorough foundation to perform such analyses. This is, hence, the necessary prerequisite to fulfill Requirement 4. How the actual integration can be realized is subject to further considerations and part of Chapter 6.

4.2 The HybridMDSD Approach

The requirements we identified in Section 3.4 demand for an explicit binding network between models that allows to qualify DSLs with additional semantic information, denoting the meaning of these languages, and to use this qualification to facilitate semantic sound references between modeling assets. In the previous section, we figured out additional technical necessities concerning the reference mechanism and appropriate technologies to facilitate required reasoning capabilities. In this section, we introduce the core idea for our solution approach as well as a brief explanation of how it shall fulfill the listed requirements.

4.2.1 Intentional Overview

The core idea is a solution approach that is entirely based on Semantic Web technologies. We want to employ a central upper ontology to capture the semantics of particular domain-specific languages. As we are aiming at the integration of languages for software specification, this ontology is called the *Unified Software Modeling Ontology (USMO)* [BL07, BL08] (see Section 6.1). To capture semantics, we map dedicated modeling constructs to the concepts and roles of this ontology. Figure 4.4 illustrates this language-level mapping. This way, each part of the DSL that serves as a candidate to establish references to different parts of the modeling landscape obtains an explicit semantic representation.

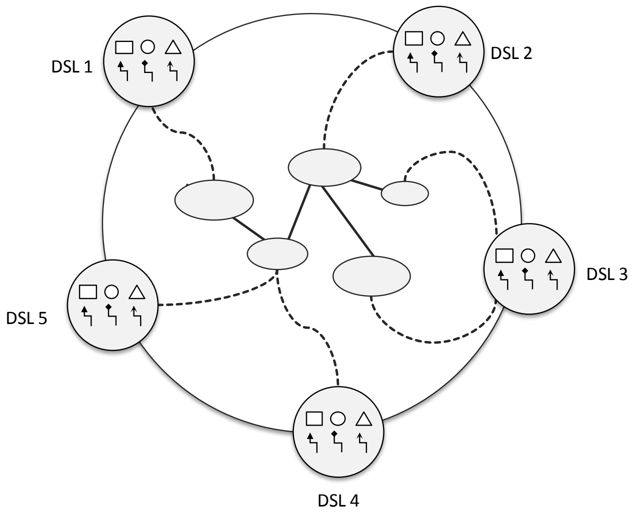


Figure 4.4: *Mapping of modeling constructs in DSLs to a central upper ontology*

The language level-mapping during the creation of actual domain-specific models is succeedingly used to build an ontology knowledgebase that instantiates previously captured conceptual information. This way, each DSM obtains a range of semantic individuals that instantiate concepts and roles derived from the software modeling ontology. These individuals represent model connection piers that can be used to establish inter-model references. Hence, the actual references are created between the semantic representation of corresponding modeling constructs. Figure 4.5 illustrates this model-level mapping.

Which references are allowed or even possible to be drawn is prescribed by the employed central upper ontology. This is because each individual maps to a corresponding ontology concept, which, in turn, comprises particular relationship definitions in the ontology. Thus, the upper ontology operates as actual *connection interface* between domain-specific languages and their instances. The ontology konwledgebase, thus, represents the required *binding network*.

We coin our solution approach *HybridMDSD*, where the term *hybrid* has different meanings. First, we combine the two different *technological spaces (TSs)* [KAB02] of MDSD and Ontologies. We leverage both, the advantages of MDSD to improve efficiency in software development and the comprehensive logics support provided by the Semantic Web to provide advanced support for consistency checks, change adaptation and guidance. Second, the term *hybrid* documents the use of different domain-specific languages to specify one single software system. A variety of semantically different DSLs is used in combination to facilitate the specification of most diverse software parts.

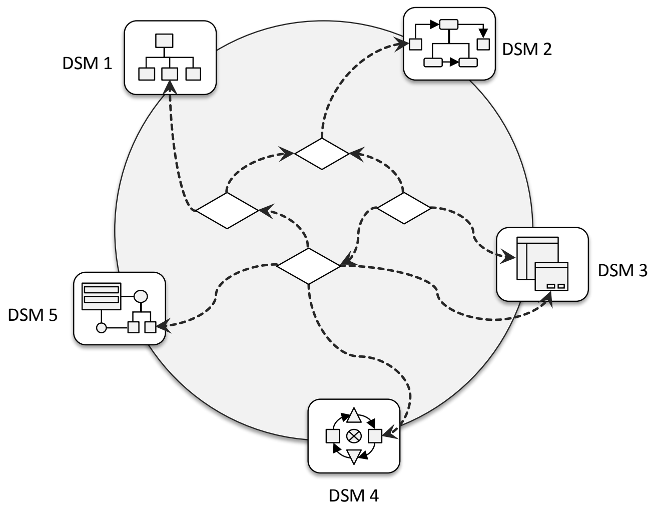


Figure 4.5: *Mapping of modeling constructs in DSMs to ontology knowledgebase individuals*

4.2.2 Architecture Overview

Figure 4.6 illustrates the HybridMDSD architecture with two exemplary domain-specific language stacks. Each of the figure's sides shows one DSL which is instantiated by several domain-specific models in a modeling landscape of a particular software product. These DSMs again form the foundation for code generation, where a corresponding generator engine produces actual programming code.

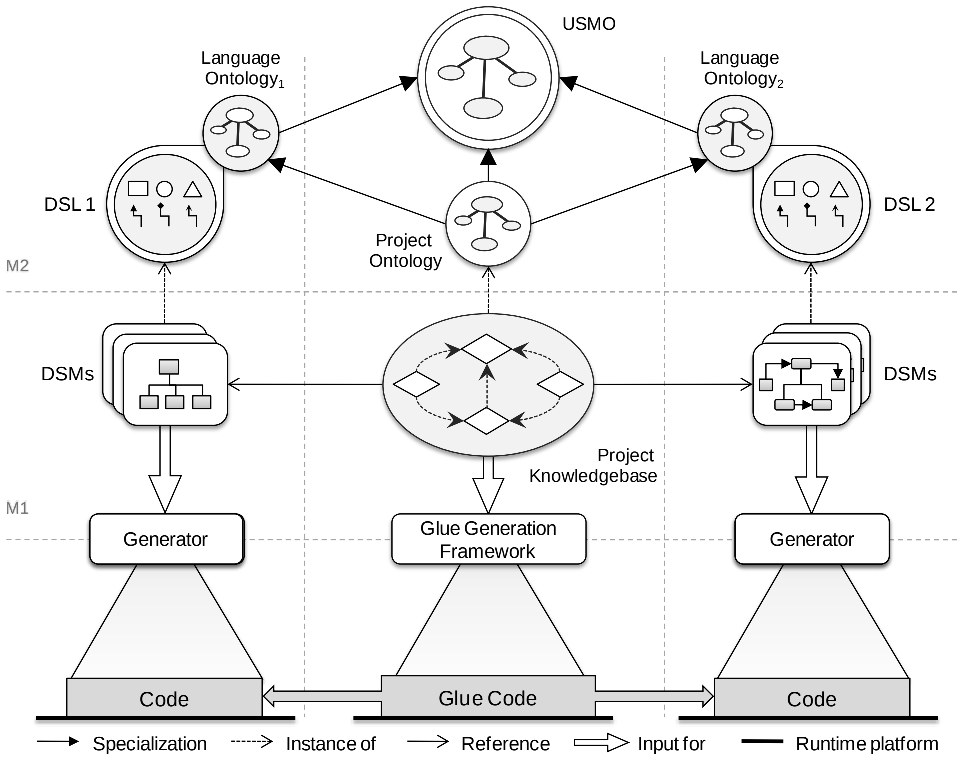


Figure 4.6: *HybridMDSD architecture illustrated with two DSLs*

4.2.2.1 Language Ontologies

For each DSL an appropriate *language ontology* is created, as illustrated with the circular region above the DSLs in Figure 4.6. The language ontologies are based on the integrating upper ontology for software models, USMO, introduced in the previous section and shown on the top of Figure 4.6. When mapping a DSL's parts to the upper ontology the *meaning* of a DSL is captured, i.e., it is interpreted in terms of the upper ontology. To this end, we identify only those modeling constructs that *commit to* the employed ontology, i.e., that have an equivalent meaning like particular parts of this ontology [Gui07]. For each committing modeling construct, corresponding ontology concepts and roles are created as direct specializations of concepts and roles of the upper ontology, as illustrated in Figure 4.7. In addition, mapping rules are developed that control the instantiation of language ontology concepts during modeling.

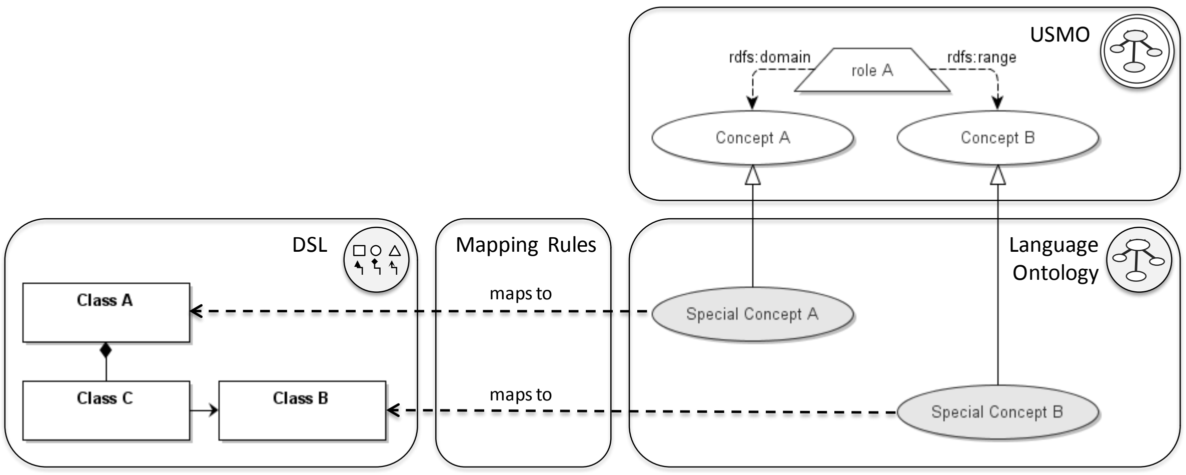


Figure 4.7: *Upper ontology, language ontologies, and mapping rules*

4.2.2.2 Mapping Rules

A language ontology and mapping rules for its instantiation are the necessary tools to automatically produce a knowledgebase during the creation of domain-specific models. According to this, the specified rules are triggered when corresponding modeling entities are created. The rules then instantiate appropriate language ontology concepts accordingly. This synchronization process is also triggered when modeling constructs disappear, as illustrated in Figure 4.8. This way, a semantic image of each DSM that conforms to the ontological commitment of its parent language is automatically instantiated. This semantic image is the foundation for further development tasks. Using the established knowledgebase, software engineers are enabled to draw connections between different DSMs of diverse languages. This process explicitly obeys the ontological foundation provided with the USMO. In other words, inter-model references can be established only on the basis of the ontological structures a particular modeling construct has created through according mapping rules. This way, created references obtain a dedicated semantics which can be leveraged in various ways (cf. Section 4.4).

4.2.2.3 Glue Code Generation

The central part of Figure 4.6 illustrates the architecture's upper ontology and the knowledgebase which is automatically created during modeling. Below those, the figure depicts the *Glue Code Generation Framework (GGF)*. This is the architectural component that is concerned with the

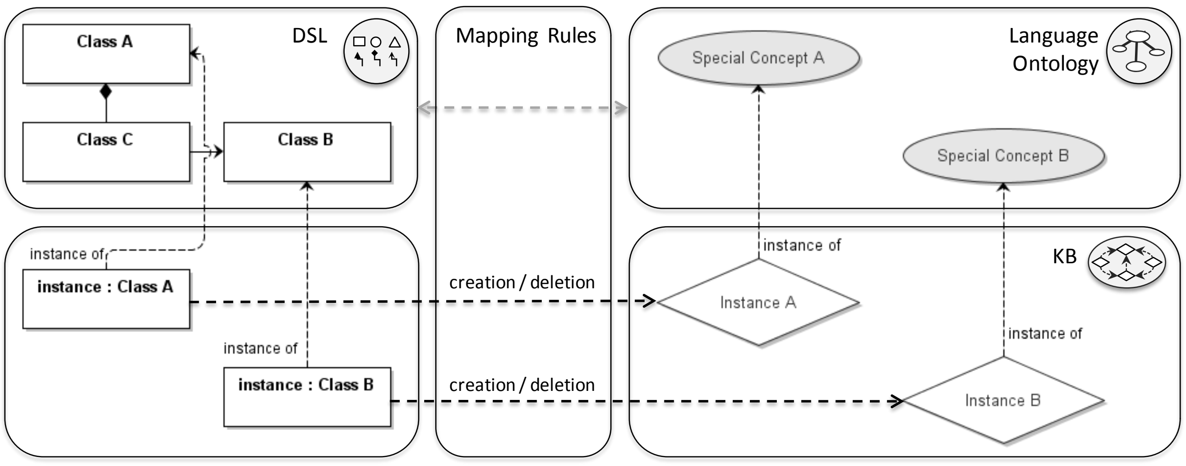


Figure 4.8: *Mapping rules for synchronization between models and knowledgebase*

integration of the execution semantics of incorporated DSLs. As illustrated, we propose a solution for generative scenarios with the HybridMDSD approach. In Section 6.2.1.2 we discuss the reasons for this and compare interpretive with generative scenarios. The GGF component is the binding interface between the connector knowledgebase and code integration structures that link generated program parts of various DSLs. To this end, the component maintains composition patterns between languages on all relevant meta layers, including M2, M1, and code level respectively. For a detailed explanation of the GGF component, refer to Section 6.2.

4.2.2.4 Language Repository

Development with HybridMDSD focuses on software products and product lines whose solution space consists of multiple DSLs. In doing so, the realizing architecture must provide appropriate means to organize and cluster different DSLs, according tools and platform-specific generators, as found out in Section 3.4. To this end, we bundle DSLs and surrounding tools in dedicated *language modules*. Each language module comprises at least the metamodel of a DSL, its language ontology, according mapping rules and various platform-specific generators. The metamodel is serialized in a format specific to a certain modeling technology and may contain additional metadata that is necessary, e.g., for particular editing tools [Kol07, Fou09e]. This way, domain-specific solution parts are condensed to coherent, self-contained modules that can be organized in a repository. Within a particular *semantic connector project*, desired languages can be searched in the repository and chosen for the development workbench. In this respect, the search process is massively improved by the language ontology of a DSL, which facilitates semantic queries based on the underlying ontological structures. Figure 4.9 illustrates the repository architecture.

4.2.3 Covered Requirements

The HybridMDSD solution approach enables us to fulfill previously listed requirements. In the following, we enumerate all requirements that were identified in Chapter 3 and explain how they are covered by our approach:

Requirement 1 The given overview did not introduce particularities about a HybridMDSD development process. This is done in detail in Section 6.3. However, already here we can state that the semantics about software models that shall be described in the employed upper ontology might be useful also for language design.

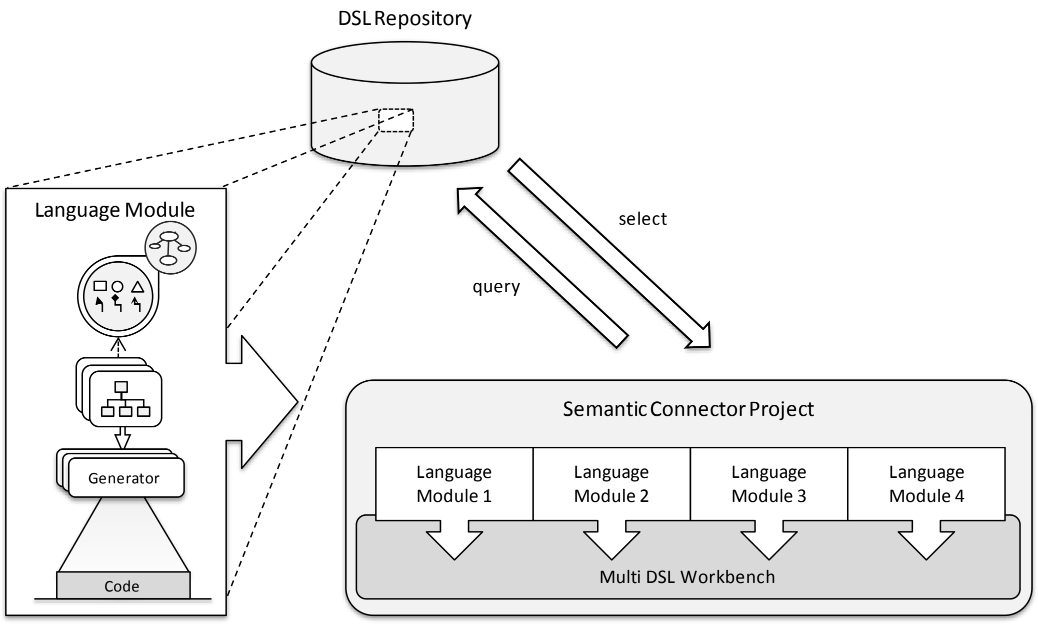


Figure 4.9: *HybridMDSD language repository*

Requirement 2 The dependencies between DSLs and references between DSMs are represented by an upper ontology and an ontology knowledgebase. Inter-language dependencies become explicit by mapping modeling constructs to ontology constructs, which are related to each other. Inter-model references become explicit through knowlegebase individuals that instantiate the language-level mapping. Using ontologies enables us to leverage advanced reasoning support for consistency checks.

Requirement 3 Through the mapping of multiple languages to one single upper ontology that is instantiated in one single knowledgebase, the demanded extensive consistency and integrity analysis is not limited to pairwise calculations.

Requirement 4 As described above, Semantic Web technologies provide a thorough foundation to perform comprehensive analyses about the referential interplay between DSLs and DSMs. This allows us also to create composition structures between DSLs.

Requirement 5 The dedicated language repository of the HybridMDSD approach organizes various modeling assets for different domains as language modules in one single place, allowing to query, contribute and use once created development artifacts.

Requirement 6 The representation of implicit language semantics is implemented by assigning particular concepts and roles to dedicated modeling constructs of DSLs. This process equips these constructs with additional information about their sense in a broader integrated scope.

Requirement 7 The upper ontology is the basis for establishing model connection interfaces. It specifies which modeling constructs of different DSLs are to be used to link different models together. Only those constructs that are relevant for such references, i.e., that have a meaning in terms of the ontology, are captured and need to be maintained.

Requirement 8 The focused employment of Semantic Web technologies forms the necessary foundation for advanced calculations that facilitate guidance and adaptation scenarios based on rule-based reasoning.

4.3 Illustrative Example

The previous sections explained the envisaged approach to implement our multi-domain engineering vision, introduced in Section 3.3. In order to provide a clearer picture of the approach, this section will demonstrate its application with a small and simple example. In the following, we illustrate the HybridMDSD approach on the basis of three domain-specific languages. We demonstrate informally how the languages interact, give insights about their mappings to corresponding language ontologies, and illustrate how different DSMs are linked together through a central project knowledgebase.

4.3.1 Scenario Overview

Assumed is a content management system that allows users to manage online surveys. On the system's web user interface, it is possible to arrange, edit and publish surveys with dedicated user interface forms. Such forms and other user interface elements are specified in a custom DSL called *Form DSL*. Pressing the submit button of a form invokes a particular service which modifies associated data structures according to the information entered in the form's widgets. Corresponding business logic is specified using the *Service DSL*. The data structures holding actual survey information are specified with the *Data DSL*. The interplay of the interacting languages is visualized in Figure 4.10.

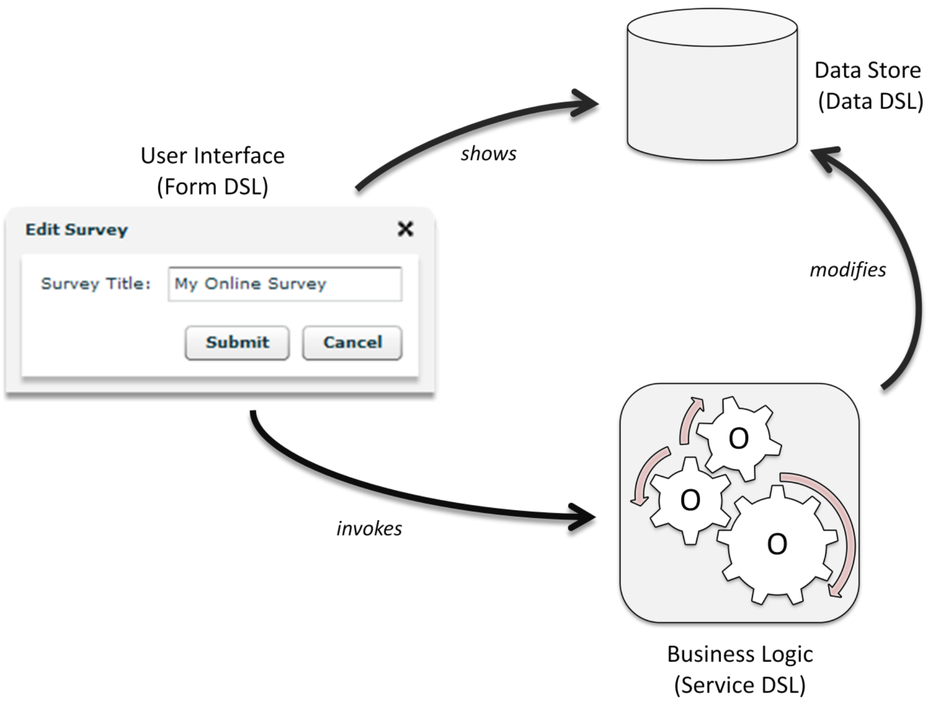


Figure 4.10: *Informal semantic connection between 3 DSLs (adapted from [LH09])*

The arrows between the diverse languages depicted in Figure 4.10 informally describe the semantics of their overlap by conceptual relations such as *shows*, *invokes*, and *modifies*. The explicit semantics of these overlaps is predefined in HybridMDSD's central upper ontology, USMO (cf. Section 4.2 and 6.1). To allow the creation of dedicated inter-model references, following the general scheme depicted in Figure 4.10, each DSL is mapped to this ontology. An exemplary excerpt of such a mapping for the Data DSL is illustrated in Figure 4.11.

The figure's left side shows parts of the language metamodel, while the figure's right side

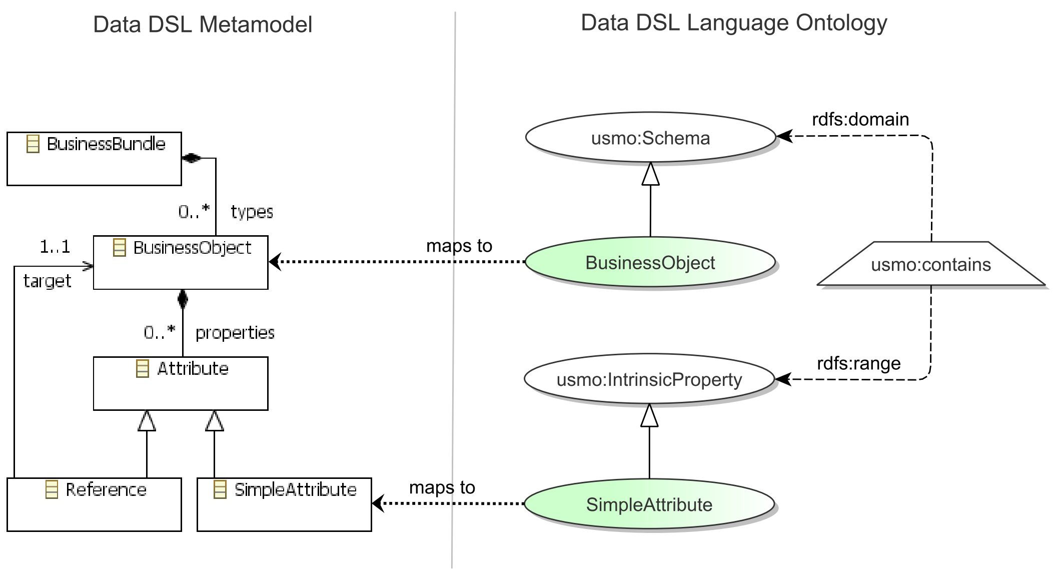


Figure 4.11: *Excerpt of the semantic mapping for the Data DSL*

illustrates appropriate green-colored ontology classes that map to these parts. As explained in Section 4.2.2.1, these classes inherit from specific ontology classes of the central upper ontology. This way, particular DSL modeling constructs obtain selected semantics in a system-wide context.

4.3.2 Modeling Landscape

Figure 4.12 depicts an exemplary modeling landscape for the above informally described scenario. It shows three models, where each is instance uses one of the introduced DSLs. The user interface model defines the EditSurvey form to edit surveys, which invokes the saveSurvey service that persists user interface changes in the associated Survey BusinessObject. Besides the different models, Figure 4.12 illustrates an excerpt of the knowledgebase that encodes different inter-model connections. According to the identified semantic mapping of each DSL, corresponding language ontology concepts are instantiated. Based on this semantic frame, semantically sound references were set by the developer.

As illustrated, user interface widgets are Representatives that *represent* other system entities, such as the buttonClick Event which triggers a the saveSurveyCall. This Action itself is an instance of a specified Behavior, the saveSurvey service, which involves instances of the Survey BusinessObject. For better clarity, the knowledgebase excerpt shows instances of original USMO classes instead of their language-specific subclasses, as illustrated in Figure 4.11. Nevertheless, the coloring shall elucidate that different concepts together with their mappings would be defined in corresponding language ontologies.

The knowledgebase of the modeling landscape can now be exploited to yield a number of advantages and development improvements. Constraints, once specified and inherited from the central ontology, can be used to assist the modeler with information about incomplete models or dangling references. For example, the title text field shown in Figure 4.10 would be useless if it would not depict a concrete data value. Corresponding constraints would inform the developer about such insufficiencies. Additional added value would, for instance, result from the production of glueing connection code between generated code from separate DSLs. Which benefits we expect in detail is described in the following section.

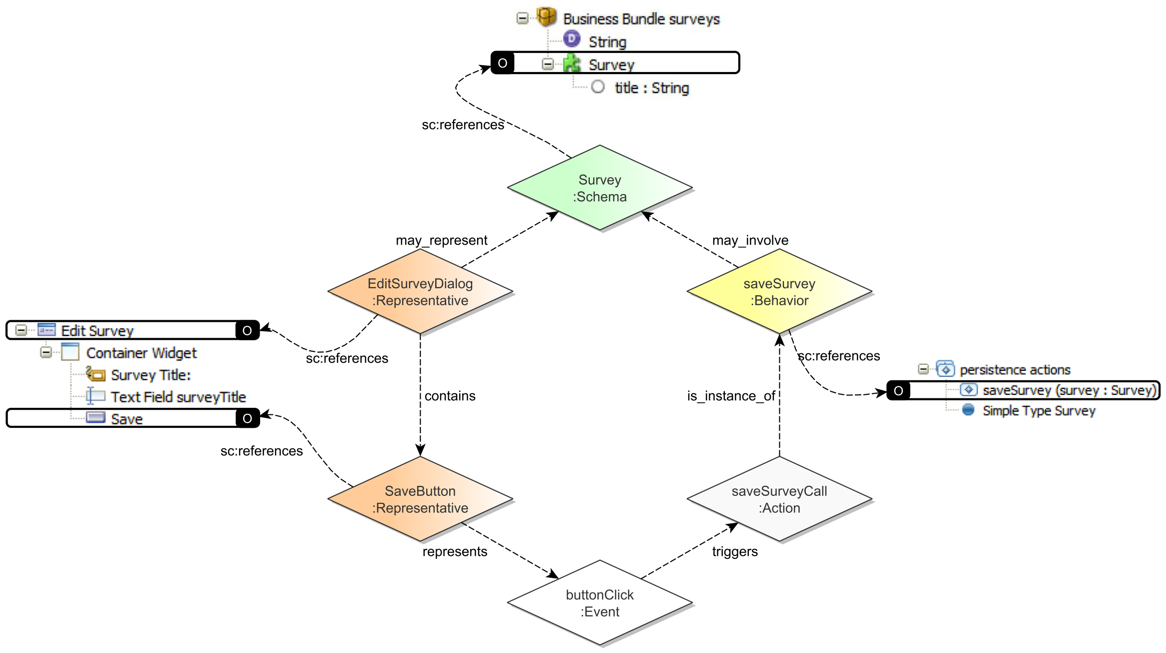


Figure 4.12: *Simplified excerpt of HybridMDSD knowlegdebase for exemplary models*

4.4 Expected Benefits

In this section we present the benefits that we expect from the HybridMDSD approach. The presentation is divided into two different areas, where we consider anticipated advantages from (1) a higher-level management perspective and (2) a lower-level software engineering perspective. In (1) we focus on economic benefits that reduce the *Total Cost of Ownership (TCO)* and facilitate efficient customer-centric development (cf. Section 4.4.1). With (2) we shed light on advantages for the software developer, who gains sophisticated support for the implementation of potentially very complex products (cf. Section 4.4.2). In this respect, the economic benefits directly arise from technological advantages.

4.4.1 Economic Benefits

4.4.1.1 Increased Value of Software Artifacts

As described by Atkinson and Kühne, model-driven software development increases the value of software artifacts by increasing the "return a company derives from its software development effort" [AK03]. According to the authors, this has two major reasons. First, the increasing short-term productivity of developers. Software developers, or modelers in MDSD, create modeling assets as primary software artifacts. These assets are higher valued because they facilitate the production of a large amount of additional assets through code generation. Second, the increasing long-term productivity of developers. The primary artifacts do not change as often as programming code, because they abstract from particular solutions implemented with specific platform technologies.

The introduced multi-domain engineering paradigm together with the HybridMDSD solution approach even increases the value of primary software artifacts by consolidating remaining volatile assets too. The approach allows not only to use multiple abstract DSLs for different purposes in parallel, but also facilitates the *maintenance of integration structures* between DSL implementations. The explicit knowledge about the interconnection of languages enables for the development of code generators that are exclusively dedicated to single languages. Generator templates must not import more than one metamodel anymore. The specified binding between inter-model connections and integrating composition structures leads to a heavily reduced rate at which code generators become obsolete. Additionally, once created, tested, and optimized generators can be employed in different project scenarios. This increases the value of all assets of a modeling landscape and, hence, the return a company can derive from its efforts to produce them.

4.4.1.2 Decreased Time-to-Market

The main ideas behind MDE and the enabling HybridMDSD approach are *separation of concerns* and *reuse* (cf. Section 3.3). Through the division of the solution space for software products into various domain-specific parts, even complex scenarios can be managed with specialized expertise of appropriate professionals. The centralized binding network based on semantic structures provides rich consistency support, which ensures that the integrity of every solution part is intact. The HybridMDSD language repository concept enables software developers to contribute their implementations to a community, such as the development collective of a company. This way, the necessary effort to setup a comprehensive development workbench for new software products is dramatically reduced. In case all needed languages and appropriate platform-specific generators are already available in the repository, development teams can start modeling from day one, with dramatically reduced efforts for preparation.

In scenarios where existing products shall be extended with additional features, the semantic foundation of HybridMDSD increases development efficiency once again. A feature's implementation potentially comprises numerous different assets that add functionality to various parts of a product. However, feature development is a sequential process, which has to be started at some place in the system. The model connection knowledgebase together with appropriate reasoner engines paves the way for automatic adaptation of software models in case of modifications. This way, software engineers are massively supported during the development of individual product features. This saves time and reduces the overall time to market.

4.4.2 Engineering Benefits

4.4.2.1 Separation of Concerns

The proposed MDE paradigm emphasizes a concern-centric separation of complex solution environments on a convenient level of abstraction. This way, domain-specific expertise can be unfolded with a maximum of efficiency and without harming technical constraints. The information that is captured in language ontologies provides explicit data about the meaning and significance of particular technical system models. This creates a necessary foundation to bridge the gap between technical and conceptual models, which was previously a major concern in the semantic community [OHS02, HH03b, Gui05, OB06, HHB+06] and is still a challenge.

Hence, HybridMDSD fosters separation of concerns in two different ways. On the one hand, language ontologies and the connector knowledgebase allow for domain specificity on a technical level. In terms of MDA (see Section 2.1.3.2), this implies a *horizontal* division of the solution space on technical layers, such as PIM and PSM. On the other hand, the explicit semantic information about DSLs supports the approximation of conceptual and technical models, such as CIMs and PIMs. This facilitates a proper *vertical* division of the solution space.

4.4.2.2 Problem-Centric Language Reuse

Through the organization of DSLs in language modules that comprise the languages' ontological commitment [Gui07] in form of an ontology, we facilitate advanced language retrieval capabilities. The Semantic Web community comes up with a rich set of tools, formats and frameworks (cf. Section 2.4). Among these, also query languages such as RDQL [Sea04] or SPARQL [PS08] can be found, which allow for requesting particular knowledge fragments from ontologies and knowledgebases. Because language ontologies in corresponding modules are amenable for such query mechanisms it becomes possible to select desired DSLs based on their particular semantics. Moreover, this even allows to effectively share and publish language modules with other stakeholders and companies online [Knu05].

For instance, a developer could easily query for a modeling language that allows to specify data type development or business logic specification to a dedicated degree of specificity. The corresponding query could be posed based on the set of appropriate concepts and roles in available language ontologies. Without the semantic information, only vague string-based queries over names of DSL classes, attributes and references could be posed, which is inaccurate and time consuming. With HybridMDSD, language retrieval can even be extended through additional metadata for code generators and target platforms. This forms an optimal grounding to find the needle in a haystack and fosters problem-centric language reuse.

4.4.2.3 Intra-Language Consistency

The employed upper ontlogy, USMO, conducts essential knowledge about software systems and their specification means (see Section 6.1). We have defined a rich set of concepts, roles, axioms, and integrity rules that comprehensively specify how different modeling entities commonly relate to each other. This information can already be used in one single language in a multi-DSL development environment. In particular, just during metamodel creation of a DSL, the knowledge about software modeling languages can be of importance to avoid integral design faults. Here, the language designer can be guided to reach his actual modeling goal.

4.4.2.4 Inter-Model Consistency

In previous sections we often highlighted the reasoning strengths of Semantic Web technologies, which pave the way for sophisticated processing over the modeling landscape. This is conducted within the here presented benefit of inter-model consistency. One major achievement of an ontology knowledgebase that interconnects different modeling assets is the capability to test and ensure integrity of these connections. This already begins with simple referential integrity, where it has to be ensured that a link between modeling entities has no dangling source or target ends. Moreover, the semantic structures of the upper ontology allow to yield more complex inconsistencies. It is, for instance, possible to report that a user interface dialog shows a data structure that is not yet furnished with any behavior specification. But this again, is probably desired to manipulate the data structure through according UI widgets. Corresponding validations are facilitated through generic and domain-specific axioms and rules (cf. Section 6.1) in the semantic core. All in all, the semantic backbone of HybridMDSD facilitates advanced development assistance that goes beyond common consistency checks.

4.4.2.5 Generation of Composition Structures

As stated in Section 4.2.2.3, the HybridMDSD approach focuses on generative MDSD scenarios. In such scenarios, domain-specific models serve as input for generator templates, which produce platform-specific programming code. In the essential requirements for MDE (see Section 3.4) and the solution approach overview given in Section 4.2.2, we highlighted the importance of reuse and maintainability for all kinds of modeling assets. Therefore, different DSLs' code generators are kept separate and the introduction of dependencies between them is strictly avoided. To still realize an end-to-end modeling approach that facilitates fully executable software applications, the language dependencies and overlaps derived from an integrating ontology are used as foundation to specify or even derive integration patterns on code level. These patterns are then used to produce code level composition structures between diverse DSLs for each inter-model connection, encoded in the central project knowledgebase. This allows a maximum of automation for the development of model-driven software applications.

4.4.2.6 Generation of Model Transformations

The inter-dependencies and relations between arbitrary DSLs are encoded in the conceptual relations provided by the central integrating upper ontology. The assigment of particular modeling constructs to appropriate ontology concepts is given by the mapping rules, which are specified for each language (cf. Section 4.2.2.2). This way, parts of different DSLs are related to each other, which is an ideal basis to automatically produce actual model-to-model transformation rules. In this respect, the semantic qualification of particular connections can additionally be of importance. For example, the information that two modeling constructs are semantically equivalent, would be the necessary precondition for bidirectional projections, while dependency relations rather indicate uni-directional transformations. Hence, once a specification about how languages interact with each other is available, this knowledge can be used to provide another piece of the puzzle for a sophisticated modeling environment.

4.4.2.7 Model Navigation

A specification of model connections, as it is available within the HybridMDSD knowledgebase, allows us to query related artifacts for each single asset in the modeling landscape. The resulting network can be leveraged to visualize dependencies and relationships between modeling assets and to provide hyperlink features for direct navigation. In an integrated development environment, like the Eclipse IDE, this navigation facility can, e.g., open relevant editors, place the cursor at correct positions in source files, or equip modeling assets with intuitive reference markers. Through available semantic information about the connections, navigation structures moreover obtain an application-specific meaning.

This yields valueable information about the significance of these connections and further assists during system modeling. If the developer, for example, selects an attribute of a data structure, we can immediately make him aware of all direct neighbors in arbitrary solution domains, such as user interface input fields in a related dialog or corresponding service methods that are used to modify the data structure. The corresponding hyperlink functionality then enables the modeler to directly switch to related artifacts. Such cross-language information is especially very useful when modelers add, modify or delete details of corresponding assets.

4.4.2.8 Traceability Support

What is useful for navigation can also be leveraged in context of traceability. The reference graph between different assets can be considered as tracing source, which gives information about the interrelation of various artifacts. Moreover, the semantic qualification of inter-model references even allows to derive a directed graph, which yields information about cause and effect releationships between assets. Such a traceability graph is useful, e.g., in software product line engineering. Here, it is essential to estimate the impact of a feature's implementation on the entire development environment, in order to improve project planning in time and staffing.

4.4.2.9 System Visualization

Last but not least, visualization is an important means to shorten rampup phases for developers and illustrate different system parts. Besides the possibility to employ graphical DSLs in the language workbench, which already visualize particular parts of a software product, the HybridMDSD knowledgebase can be used as source for advanced visualization tools, such as interactive 3D representations or animated graph structures [oD09]. Visualization features of a language workbench seamlessly integrate with above described benefits. For instance, navigation support could be accessible from a global visualization of the software core. Inconsistencies could be presented with distinct colors and traceability information could be rendered differently.

Chapter 5

Related Work

In Chapter 3, we comprehensively analysed actual problems that challenge IT companies in software development. To alleviate these problems, we sketched our vision of multi-domain engineering, which facilitates and improves model-driven development with multiple domain-specific languages in variability-intensive scenarios. In Chapter 4, we revealed a first overview of the realizing approach for this vision, HybridMDSD. Before presenting the results of our research work in Chapter 6, this chapter summarizes important projects that are related to HybridMDSD. As the MDE vision involves a number of technologies and development paradigms, our related work analysis covers different research areas.

First, MDE and the proposed HybridMDSD approach offer an end-to-end framework for model-driven software development with multiple DSLs. We envisage purely model-driven development that leads to executable applications. Consequently, Section 5.1 reviews *comprehensive* approaches that try to realize such end-to-end scenarios likewise. Second, the core idea of our approach comprises a powerful method to implement model composition and to leverage resulting advantages. What important alternative ways to compose models of different DSLs exist and how they differ from HybridMDSD, is discussed in Section 5.2. Last, our solution approach utilizes semantic foundations amenable to reasoners that allow for logical deduction in the context of software engineering. Therefore, we review important research work which follow comparative approaches in Section 5.3.

5.1 Comprehensive Approaches to Multi-DSL Development

In this section, related approaches to an end-to-end solution for multi-DSL development are introduced. To this end, it is important to highlight that much work has been done that either considers particular elements of such a solution, like the important task of model composition (see Section 5.2), or that leads more general discussions on semantics-based solution approaches, as presented in Section 5.3. Comprehensive approaches that provide a solution that covers scenarios envisioned by our work, i.e., multi-DSL scenarios in a variability-driven context that shall end in executable applications, are rather hard to find. In the sequel, we present standards and projects that are most narrow to HybridMDSD even if they not necessarily cover all the requirements of our approach.

5.1.1 The OMG Approaches UML and xUML

With the *Unified Modeling Language (UML)*, the OMG took a big and important step towards model-driven software development. Although the specification is rather old — first discussions took place before the turn of the millennium [RP97, Qua03] — it was one of the first remarkable joint efforts of industry representatives to facilitate the model-driven specification of software from multiple viewpoints. UML allows to describe a multitude of software facets with different diagram types that enable developers for both structural and behavioral modeling. Prominent examples are class, component, or object diagrams, for structural modeling, and sequence, state machine, or use case diagrams, for behavioral modeling [UML07]. In this respect, from a software architecture point of view UML provides multiple domain-specific languages to model software.

Figure 5.1 depicts the semantic areas covered by the different UML notations. The figure shows three layers, where each layer forms the foundation for superior layers. The bottommost layer is of structural nature, which reflects the premise of UML that all behavior is the consequence of actions between structural entities.

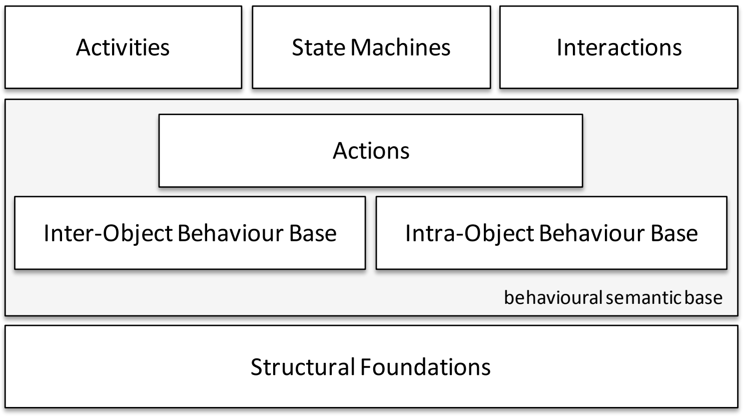

Activities
State Machines
Interactions
Actions
Inter-Object Behaviour Base
Intra-Object Behaviour Base
behavioural semantic base
Structural Foundations

Figure 5.1: *UML semantic areas and their dependencies (adapted from [UML07, p. 11])*

The UML was developed "with a very broad scope that covers a large and diverse set of application domains" [UML07, p. 1] and acts as *general purpose modeling language (GPML)*. This resulted in a highly complex specification and rendered UML as barely suitable for particular domain-specific use cases. To achieve more flexibility and also support domain specificity, appropriate means were introduced, such as tagged values, stereotypes and constraints [OMG01, pp. 67-77].

Furthermore, the UML specification does not explain how to derive executable software applications from corresponding models. This was also witnessed by the research community [Mor01, MBB02, FGB04] and led to efforts towards *Executable UML (xUML)*. The main goal of xUML is to equip a subset of the conventional UML 2 metamodel with "a precise definition of the execution semantics" [Gro08, p. 5] in order to facilitate the "construction, verification, translation, and execution of computationally complete executable models" [Gro05, p. 1]. While its specification was officially initiated with a *request for proposals (RFP)* in 2005, the current xUML specification is in beta state termed *fUML (Foundational UML Subset)*.

The Unified Modeling Language has a number of benefits. It is an holistic approach to model different software facets with dedicated diagram types, or DSLs. It provides the means to extend default modeling primitives with different extensibility mechanisms to achieve domain-specialization. Finally, with xUML it allows to specify computationally complete executable models that facilitate automated derivation of fully executable platform-specific applications. In total, this renders the UML together with xUML a comprehensive approach to multi-DSL development, which justifies its consideration in the context of this chapter. However, there are also a number of reasons why UML can not sufficiently implement the MDE approach as proposed in Chapter 3.

First, UML models rather seldomly form a consistent modeling landscape. A great number of CASE tools support development with UML [FTSDT09, Tea09, IBM09, NM09, Ltd09]. Such tools usually do not enforce inter-model constraints, which is advantageous in early analysis phases and makes it easy to start specifications from scratch. However, the freedom to model without ensuring, e.g., referential integrity between class and statechart diagrams, leads to inconsistencies across models and entails potential problems in subsequent development stages [LC04]. This is one reason why UML notations are often used as documentation assets only.

Next, the extension mechanisms UML provides can indeed help to establish domain-specific layers on top of usual diagram types [LMTS02, AE02, PF02]. Anyhow, when constructing such domain-specific views, a language designer is oblidged to start from existing UML notations. This might not always be appropriate, for example, if the semantics of particular model elements needs to be entirely redefined to fit into certain domains or the semantics of two existing model elements needs to be used in combination. Furthermore, in case only very small languages shall be developed, DSL designers are obligded to massively restrict the very complex UML base languages instead of being able to start development from scratch.

Last, the attractive vision of executable UML models with xUML sounds very promising. Although it was previously criticized [Amb02] xUML seems to be more current than ever, especially in times where cloud and parallel computing becomes more and more popular [Sta08, Sta09]. However, the executability clearly comes at the price of expressiveness. Only a subset of UML is reflected in the xUML standard. This renders the approach, in addition to the rather moderate domain-specialization capabilities of UML, even less practicable for concrete domain-specific use cases.

5.1.2 Software Factories

In [GS04], Jack Greenfield and Keith Short document the industrialization process in software development. The autors compare industrialization in production of physical goods to the automation efforts in development of software. They discuss a number of “critical innovations” that need to be integrated in order to “define a highly automated approach to software development” [GS04, p. 161]. In this respect, Greenfield and Short coined the notion of *software factories* that are defined as follows:

> “A software factory is a software product line that configures extensible tools, processes, and content using a software factory template based on a software factory schema to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring framework-based components.” [GS04, p. 163]

The authors describe how to develop software product lines with various different models that describe software from different perspectives and different levels of abstraction. A software factory is mainly built of two major parts, a *software factory schema* and *software factory templates*. A software factory schema describes the product line architecture by specifying what DSLs shall be used and how models based on these DSLs can be transformed into other models, code or other artifacts. In this respect, such a schema is described as a “directed graph whose nodes are viewpoints and whose edges are computable relationships between viewpoints called mappings” [GS04, p. 166].

A software factory template provides the assets that were previously described by the underlying schema. Among others, this comprises patterns, editing tools, frameworks or samples. Figure 5.2 illustrates a simple software factory schema. Therein, rectangles represent viewpoints, dashed lines represent refinements along different levels of abstraction, and solid lines represent constraints between different viewpoint domains on the same level of abstraction.

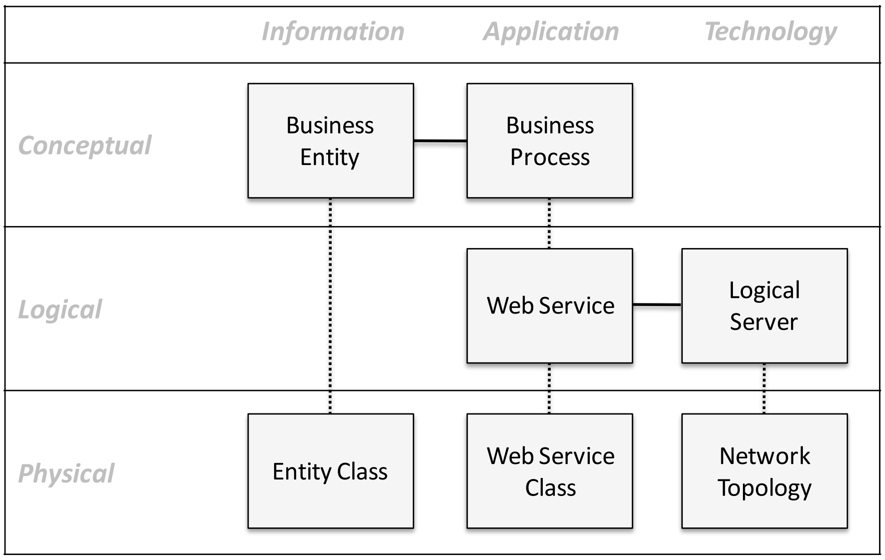


Figure 5.2: *A simple software factory schema (adapted from [GS04, p. 180]*

In summary, Greenfield and Short are aiming at an integrated development of software product lines from multiple domain-specific viewpoints. They base their consideration on critical innovations with a clear focus on a raised level of abstraction, systematic reuse and the employment of domain-specific languages. The authors highlight the importance of different perspectives in software development as well as means to ensure consistency between these perspectives [GS04, p. 169]. Hence, the main goals and the general approach of software factories, are very related to what we try to achieve with the work presented in this thesis. However, there are also at least two main differences.

First and foremost, the specification of relationships between domain-specific development viewpoints is limited to the description of mappings that are implemented by transformations only. In this respect, all types of transformations are taken into account, including model-to-model and model-to-code tranformations. However, the very important topic of expressing relationships between diverse DSLs in a sofware factory is broken down to technical specifications of pairwise transformations. This is not in line with Requirement 2 and especially Requirement 3 which demands relationships to facilitate the integration of multiple language at a time (see Section 3.4.2). This renders the whole approach rather fragile because direct transformations are heavily bound to the metamodels of source and target languages. If, for example, either the source or the target metamodel underlies requirement-driven modifications, a transformation between corresponding languages becomes obsolete and must be adapted.

Second, regarding the actual integration of DSL implementations, the software factories approach leaves many questions unanswered. The authors limit their explanations to a collection of different alternatives during the transformation process, including *horizontal*, *vertical* and *oblique* transformations [GS04, p. 456] (cf. Section 2.1.2.4). Furthermore, they explain how to solve problems when composing user-written with generated programming code [GS04, p. 477]. Nevertheless, a dedicated approach to derive an executable software application from software factory viewpoints is not given, which we count to our goals as documented in Requirement 4.

5.1.3 Web Application Frameworks

Especially in the area of web application engineering multi-DSL development is well-known for a long time. High performance requirements and rather complex use cases of web-based enterprise solutions led to the development of huge platforms, such as the *Java 2 Enterprise Edition (J2EE)* [FC05, Mic09a]. While J2EE standardized many important middleware concepts, it provides, for example, a standard interface for distributed transaction management, directory services, and messaging, application programming support was rather inadequate [Joh05]. As a response, predominantly the open source community started to develop frameworks that aimed at a simplified use of J2EE. With *Struts* [Fou09b], the *Apache Software Foundation* [Fou09a] firstly elaborated on the deficiencies of the user interface technology *Java Server Pages (JSP)* [Mic09c] with their own Front Controller [Joh05]. Later, advanced solutions for persistency mappings emerged with *Hibernate* [hT09].

Today, several web application frameworks exist, where some already identified the various necessary descriptors and configuration files as ideal candidates for domain-specific languages. In the following we introduce two of them that heavily employ multiple DSLs.

5.1.3.1 JBoss and Seam

The *JBoss Application Server (JBoss AS)* [JBo08] in combination with *JBoss Seam* [YH07, JBS08] provides a certified Java EE platform for developing enterprise Java applications. In this respect, JBoss Seam enables for seamless integration of different DSLs for user interfaces, webpage navigation, session contexts, business rules, and persistence. While the actual business logic is still implemented in pure Java, it can be easily connected to assets of other application parts through string-based identifiers. To this end, JBoss Seam makes heavy use of Java annotations that are parameterized with these identifiers. Figure 5.3 shows the Eclipse-based editor for the *Java Process Definition Language (jPDL)* [RHM09], the DSL to describe user interface navigation rules.

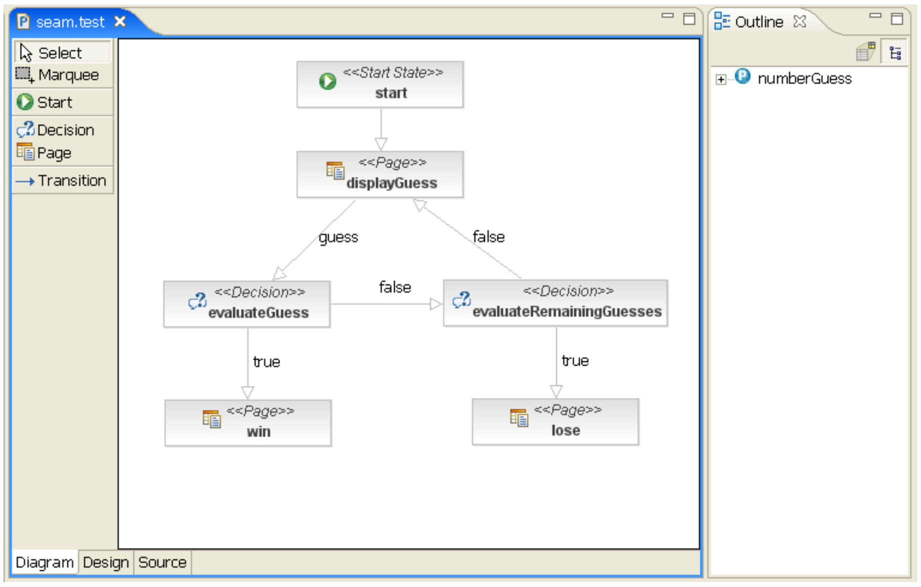


Figure 5.3: *Exemplary jPDL page flow in Eclipse editor (from [KMR+08, p. 184])*

5.1.3.2 Apache OFBiz

With the *Apache Open for Business Project (OFBiz)* [Fou09d] the Apache Software Foundation provides an open source platform for building enterprise automation software around ERP, CRM, and E-Commerce. The framework is used by both, large companies and small and medium-sized ones [HCW07]. Technically, OFBiz is a J2EE framework that facilitates business application development with up to 17 different DSLs to express various application facets, such as persistent business objects, business rules, workflows, rule-based security, or localization [HCW07]. Table 5.1 summarizes available DSLs. As an interpretive system (cf. Section 2.2.3.1), OFBiz is completely independent from executing platforms. This even counts for the business logic implementation, which is also described in an abstract fashion based on a DSL called *Minilang* [Jon04, WH08]. As in JBoss and Seam development assets are integrated through string-based references.

Tier	DSL	Description	No. of Elements
Data	Entity Model	Define business objects, attributes, and relations	23
	Fieldtype Model	Define attribute types	3
	Entity Config	Configure data sources, files, and transactions	19
	Entity Group	Configure active models and entities	2
	Entity ECA	Define events, conditions, and actions for entities	5
Service	Service Definition	Define service interfaces and permissions	18
	Service Group	Configure active models and services	3
	Service Config	Configure security, threading, and service engine	13
	Service ECA	Define events, conditions, and actions for services	6
	Minilang	Implement services	154
	XPDL	Define workflows	89
UI	Screen	Implement screens and layout	65
	Form	Implement user forms and data binding	57
	Menu	Implement menus	27
	Tree	Implement visual tree structures and data	38
WWW	Site Config	Define web controller behaviour	15
	Regions Definition	Define screen regions	3

Table 5.1: *Overview of the OFBiz DSLs (adapted from [HCW07])*

5.1.3.3 Summary and Conclusion

JBoss with Seam and the OFBiz framework are two good examples for state of the art, industrial strength in multi-DSL development. The frameworks conveniently integrate different DSLs and partially provide dedicated tooling for intuitive development. This simplifies the rather complex task to specify software in the area of enterprise web applications. However, although such solutions might be suitable for this restricted application domain, their broad applicability is rather unrealistic but covered by our MDE approach (see Section 3.3.1).

As indicated above, web application frameworks are interpretive systems that circumvent many inconvenient questions which occur when trying to combine different DSLs. Because implementing platforms know about employable languages, there is no need for flexible connection structures between language semantics. Instead, fixed and highly optimized implementations handle the control and data flow between various assets. Of course, these platforms are performance consuming and demand large-scale server infrastructures. Additionally, through the hard-coded platform implementation, given frameworks prescribe the number and nature of applicable languages. Thus, there is no possibility to extend the set of available DSLs or to customize languages. Last but not least, the prominent use of string-based references in the modeling landscape implies a number of problems concering maintainability and error-safety [HS08, HW08].

Taking a closer look at the evolution of web application frameworks yields an interesting pattern. The complexity of existing standards caused the need for simplification in development. This in turn, brought out several development tools that are specialized towards dedicated domains in the engineering process — domain-specific web application languages were born. Questioning why such an evolution has not taken place yet in other application domains, like, e.g., the domain of desktop applications, reveals one major particularity in web application development. Especially the distributed infrastructure of such applications forced software vendors to think about holistic solutions that take care of important qualities, such as performance, security, or scalability. This resulted in a rather fixed set of requirements for the entire domain, which was the necessary foundation to put financial efforts into platform development. Other application domains, however, do not have this requirement scope. The freedom to develop arbitrary sofware applications with arbitrary components, frameworks, patterns, etc., makes it difficult to agree on a fixed set of languages for development.

This is why the set of available DSLs must be flexibly adaptable for particular use cases. Each project has dedicated requirements driven by a multitude of different interests and stakeholders. A comprehensive approach to realize MDE must, hence, support not only development with fixed sets of languages, but rather provide tooling for developing new and assembling foreign languages. This is implemented by the HybridMDSD approach, introduced in Chapter 4 and explained in detail in Chapter 6.

5.2 Approaches to Model Composition

One of the most important features of HybridMDSD is the possibility to combine different DSLs with each other. The approach targets loose integration of languages and models based on a central ontology and connector knowledgebase (cf. Section 4.2). This increases the value of modeling assets because it improves reuse capabilities and helps to efficiently define arbitrary language workbenches for various application scenarios.

There are several other approaches in related work that enable for language integration. In this section, we will review the most important alternatives and analyze them for strengths and weaknesses compared to HybridMDSD.

5.2.1 Model Transformations

As introduced in Section 2.1.2.4, model transformations are an important element of model-driven software development. They allow to transform models from one representation into another, while bridging different levels of abstraction or even crossing domain borders through language migration (cf. Table 2.1).

In multi-DSL scenarios, model transformations can be used as means to compose various languages by providing bi-directional rules that map all languages in a modeling landscape to each other. With declarative approaches, like QVT relations (see Section 2.1.2.4), even change-propagation is possible, where the modification of one model triggers the execution of corresponding transformation rules that synchronize the modification in adjacent models [QVT07, p. 18]. Figure 5.4 illustrates a multi-DSL scenario implemented with model transformations.

However, the employment of model transformations in a multi-language scenario also has a number of serious drawbacks. Transformation rules do not explicitly state information about

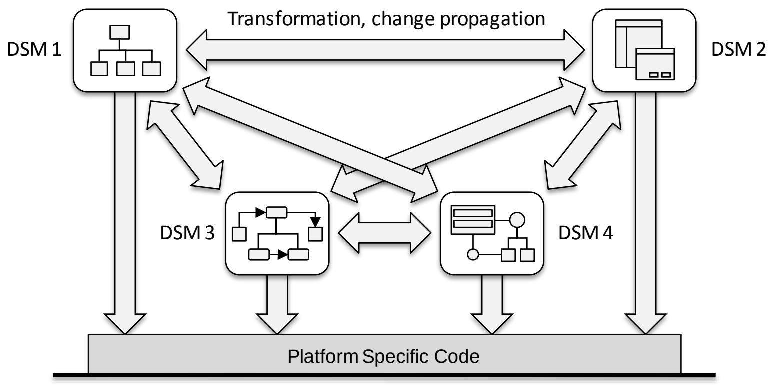


Figure 5.4: *Model transformations for multi-DSL development*

the *meaning* of modeling constructs. Indeed, they describe equivalence relationships between languages, but they do not expose what semantics such modeling constructs represent in a system-wide context. Instead, the meaning of these constructs is only implicitly available and is not revealed before interpreting or executing at least one of the involved languages. Both, the weak equivalence relations and their implicit encoding hamper the potential of model transformations for further processing. Inter-language information available from transformation rules is not expressive enough to be used for advanced tasks, such as the integration of languages' execution semantics. A better solution is the explicit representation of modeling-construct semantics, as it is demanded in RequiremEmt 6.

Another weak point of model transformations in multi-DSL scenarios is that the relational knowledge between languages is strictly bound to involved metamodels, which seriously harms reusability. For example, a rule that specifies relations between one source language and two other target languages can only be used in exactly this constellation. If one of the involed metamodels is unavailable, e.g., in a different project where not all but only several DSLs shall be reused, the entire rule becomes obsolete and must be respecified. The centralized encoding of inter-language relationships, as demanded in Requirement 2 and 3 offers promising alternatives. This higher-level specification could, for instance, be used to derive model transformations (see [RB06] and Section 4.4.2.6) instead of using them as first class inter-language relation construct.

Last, comparable to what was already explained when reviewing the software factories approach in Section 5.1.2, the strict metamodel binding of transformation rules is harmful in case of DSL modification. Changing only one of the involved metamodels breaks existing transformations, which is problematic when DSLs evolve. Of course, also a centralized approach needs to maintain references to metamodel constructs and is not independent of metamodel changes. Nevertheless, the restriction to absolutely necessary language relations together with a light-weight loose coupling approach, e.g., based on unique element identifiers, might alleviate the problems of language evolution. This was demanded in Requirement 7 and further scoped in Section 4.1.2.

5.2.2 Model Weaving

The idea to capture correspondencies between metamodels in explicit representations already came up in 2001. So-called *linkage models* were proposed to explicitly specify the relationships between entities of different metamodels [SOEB01]. Today, the probably better known term for this task is *model weaving*, coined by a sub project of the AMMA Platform (see Section 2.1.2.5), the *Aspect Model Weaver (AMW)* [FBJ+05].

Model weaving allows to explicitly create semantic relationships between DSLs, based on so-called *weaving models*. Weaving models adhere to a central core weaving metamodel or specializing *domain-specific weaving metamodels (DSWMs)* and define relations between metamodels of two or more languages. To this end, DSWMs provide different relationship semantics with dedicated links types, such as *equality* [FBV06]. Weaving models, in turn, instantiate these link types to establish actual language connections. Each link is composed of several link ends that point to particular metamodel entities. Figure 5.5 illustrates the use of weaving models in a multi-DSL scenario.

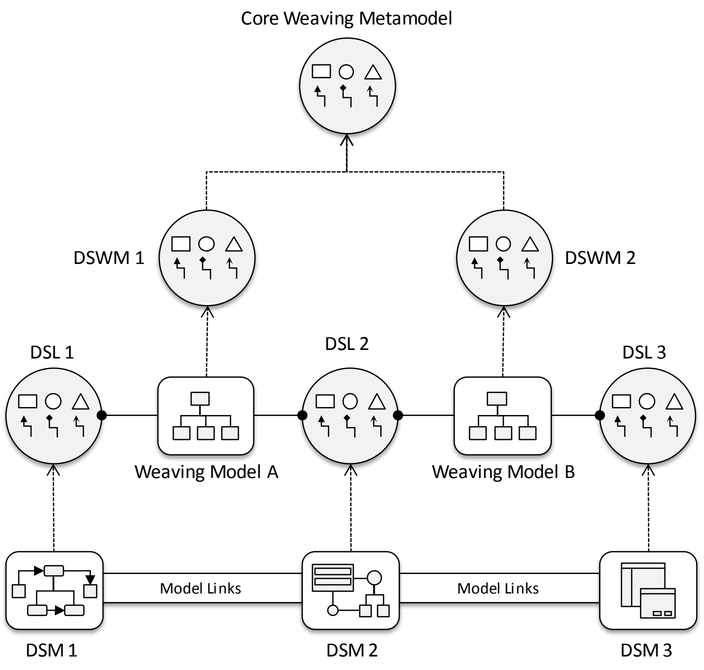


Figure 5.5: *Model weaving for multi-DSL development*

Weaving models have been proposed for several different scenarios, including data model management, tool interoperability, traceability or model alignment [FBJV05, FBV06]. Predominantly the mature tool support — AMW is provided as an Eclipse incubation project — allows to efficiently apply the concept in various contexts. Furthermore, weaving models facilitate the derivation of model transformations from explicitly available inter-language relations. It has even been shown that the process of creating weaving models can be performed semi-automatically with the help of model matching strategies [FV07].

However, model weaving also has limitations. First of all, weaving models allow to specify the semantics of language connections based on *links* only. This restricts the applicability of weaving models to scenarios with languages that are semantically narrow. If, for instance, two DSLs of entirely different domains need to be connected, shallow links might not be appropriate anymore because the semantics of both DSLs is so different that more complex constructs are necessary to describe their correspondence.

Another big limitation is that weaving models can be applied on *one* metalevel only. In several examples [FBJ+05, FBJV05], the approach is used to create inter-language relations. To this end, also domain-specific extensions can be used that narrow the relation semantics, as described above. However, this language-level mapping can not be instantiated. The information about entity relationships between metamodels is static and fixed and can not be used as some kind of reference interface for particular models afterwards. On the other hand, in cases where weaving models are applied on M1, a proper foundation on language level is missing. This renders the whole approach rather impracticable for comprehensive MDE scenarios, as it is envisioned within this thesis. Our HybridMDSD approach, instead, provides means to bind DSLs on both levels of abstraction, M2 *and* M1, while maintaining a direct relationship between these levels.

5.2.3 Semantics-based Approaches

In literature, the potential of computational logics for the integration of languages and models in MDSD has already been identified. Concerning the core idea of HybridMDSD (see Section 4.2.1), such approaches are very narrow to what we propose. In the following we review and discuss interesting projects that also employ semantic foundations.

5.2.3.1 The SemIDE Proposal

In [BR06], Bauer and Roser presented their vision of *semantic-enabled Software Engineering and Development (seSED)*. The authors focus on a development environment that allows "to combine the MDSD approach with ontological concepts" [BR06, p. 2] through a broad application of ontologies. To this end, they introduce their proposal of a *semantic-enabled Integrated Development Environment (SemIDE)* which is shown in Figure 5.6.

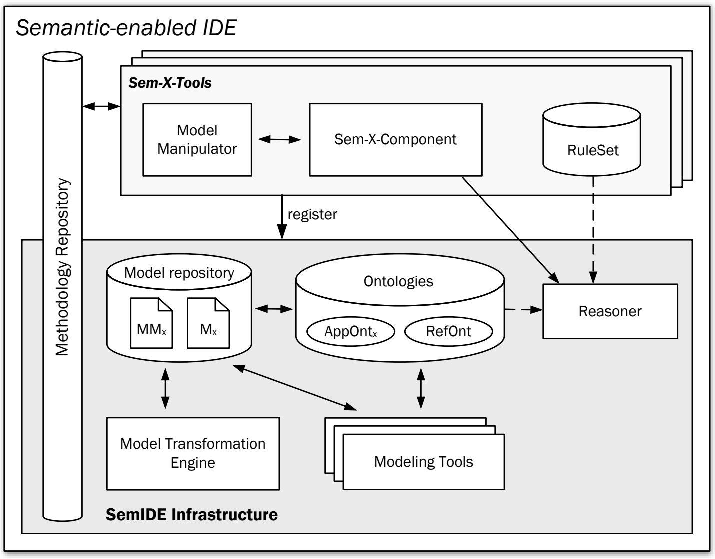


Figure 5.6: *Architecture of a semantic-enabled IDE (adapted from [Brä07a])*

The basic concept of SemIDE is to maintain ontological representations for all kinds of modeling assets, such as metamodels of DSLs (MM_x) and actual models (M_x). This is achieved by generating appropriate application ontologies ($AppOnt_x$) for these assets and synchronizing them through so-called "bridges" [BR06, p. 6]. A set of application ontologies may be bound to appropriate reference ontologies ($RefOnt$) for dedicated domains, which helps to gain interoperability between metamodels of that domain. Several plugins for the proposed platform (*Sem-X-Tools*) can then be applied to leverage derived information, available from the Ontology Technological Space [KAB02]. Here, they aim at advanced consistency checks and improved interoperability between metamodels.

At a first glance, both the intended achievements and the enabling mechanisms, are strongly related to what we propose with HybridMDSD in Chapter 4, but the approach also differs in a number of points. The application ontologies are direct reflections of (meta-)models which explains why they can be generated automatically. Actual *additional* semantic information about modeling assets is derived only when the application ontologies are bound to appropriate reference ontologies which still is a manual process. Now, the employment of several reference ontologies, each for a different domain, does not enable for interoperability between *different* domains. In HybridMDSD, instead, we use a central upper ontology that defines semantics shared between different technical domains. This also facilitates connections between most diverse heterogeneous DSLs and models.

Additionally, the authors do not estimate the necessary expressiveness of mentioned reference ontologies but rather assume their general availability. In a related article [RB06], they demonstrate an exemplary reference ontology for the domain of process modeling. With their *semantic-enabled model transformation tool (Sem-MT-Tool)* they prove the feasiblity for the generation of model transformations between metamodels mapped to this reference ontology. However, this is only one use case for a single domain and is not intended to allow for general application.

5.2.3.2 ModelCVS

The *ModelCVS* project [JV09] is a joint venture of Vienna University of Technology, the Johannes Kepler University Linz, the Austrian Ministry of Defense and an industrial partner. The main goal of ModelCVS is to achieve the integration of diverse modeling tools, by providing means for transparent transformation between models of different MOF-based modeling languages and advanced versioning capabilities for concurrent development in MDSD [KKK+06, KKR+06]. To this end, the approach allows to derive so-called *bridging operators* that facilitate the generation of actual model transformation code, which enables to seamlessly transform models between different metamodels.

ModelCVS actually shares many concepts with the previously introduced SemIDE proposal (see Section 5.2.3.1) but employs a slightly different terminology. Both approaches use ontologies to represent the semantics of metamodels. What is called application ontology ($AppOnt_x$) in SemIDE, is termed *tool ontologies* in ModelCVS. While the process of creating these ontologies is explicitly called *metamodel lifting* by the ModelCVS team [KKK+06, p. 6], Bauer and Roser here simply talk about "generic mechanisms" [BR06, p. 6] using concepts described in [ODM06]. Both approaches need to "bind" ontological representations for metamodels to either reference ontologies ($RefOnt$) or *generic ontologies* (see Figure 5.7). Even the projects exemplary use cases seem to be related, at least for the model transformation scenario. Both of them try to improve the interoperability between workflow DSLs in the domain of business process modeling. In the ModelCVS case study, an integration between metamodels of *AllFusion Gen* (MM_{Gen}), *UML Activity Diagrams* (MM_{UML-AD}), and the *Business Process Execution Language (BPEL)* (MM_{BPEL}) is achieved. In [RB06], the SemIDE platform is used to generate transformations between anonymous process modeling metamodels, using OWL-S [MBH+04].

Besides several commonalities to the HybridMDSD approach, ModelCVS also differs. First of all, the project focuses on tool integration, which is entirely out of scope in our approach. Multi-DSL development is considered as "multi-tool" development, where different tools not necessarily contribute to the same software project. Instead, synergies between different projects shall be fostered by sharing the data about tool integration in a central knowledgebase.

The most important difference to our approach is that ModelCVS entirely concentrates on language-level integration. The prototype enables to derive bridging operators based on metamodel structures, that succeedingly allow to transform models from one language into another.

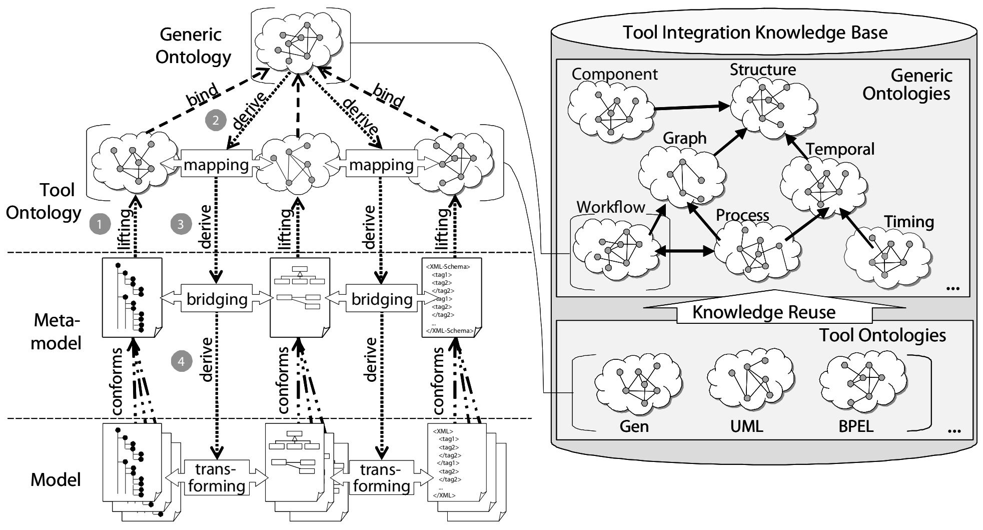


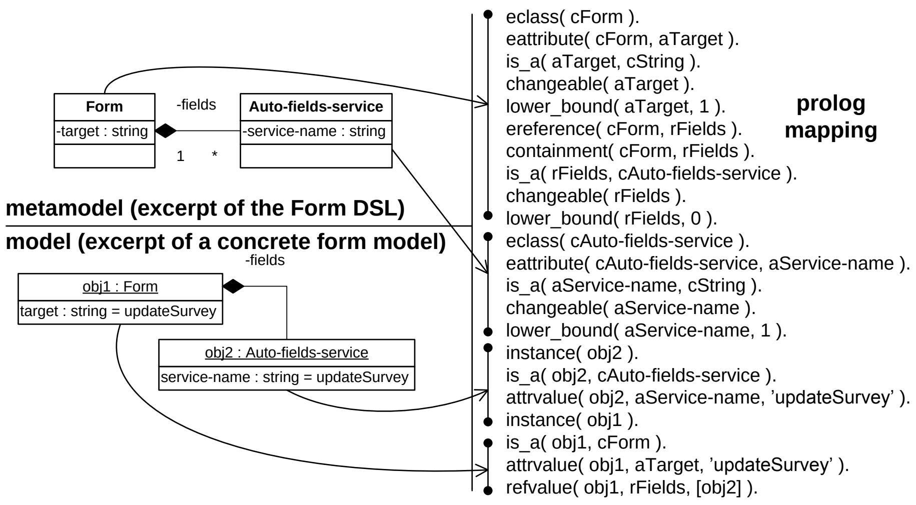
Figure 5.7: *Ontology-based metamodel integration in ModelCVS (from [KKK+06])*

What is completely missing is support for consistent inter-language modeling. To this end, we propose a central knowledgebase that yields ontology individuals for each relevant modeling construct of each employed DSL (see Section 4.2). This enables us to provide full instance-level reasoning capabilities in the *ABox* (see Section 2.4).

5.2.3.3 SmartEMF

Anders Hessellund created an extension to EMF that provides means to represent, check and maintain constraints between modeling artifacts [HCW07]. The project is based on Prolog [CM03], which enables for constraint checking. He coined the extension *SmartEMF* [Hes09], to document the contributed support for logics processing in EMF. The core idea of SmartEMF is to generate and maintain logical representations of EMF models in parallel to these models. The logical representation is formalized in Prolog and yields a holistic fact base of the modeling environment for both metamodels and models. Figure 5.8 shows an excerpt of such a mapping for a simple DSL for data structures and an exemplary instance model. Based on the logical representation, additional constraints can be specified for intra- and inter-model consistency checks. During modeling, SmartEMF provides support to perform these checks and guide the modeler, e.g., by proposing valid values for particular references.

Now, HybridMDSD and SmartEMF share most importantly the overall project goals. Both approaches are aiming at enabling consistent multi-DSL development. In this respect, Hessellund presented a case study [HCW07] based on the previously introduced OFBiz framework (see Section 5.1.3) and demonstrated how SmartEMF supports multi-modeling with string-based soft references. One major difference is the applied technological foundation. While HybridMDSD employs ontologies and knowledgebase individuals, SmartEMF is limited to Prolog facts. This entails shortcomings when trying to specify general software system semantics. Prolog does not distinguish between types and instances, like it is possible, e.g., in OWL. Hence, with Prolog it is at best possible to establish generic rules, instead of being able to provide a comprehensive type-level description for the semantics of dedicated domains in software development.


Form
-target : string
-fields
1
*
Auto-fields-service
-service-name : string
metamodel (excerpt of the Form DSL)
model (excerpt of a concrete form model)
-fields
obj1 : Form
target : string = updateSurvey
obj2 : Auto-fields-service
service-name : string = updateSurvey
prolog mapping

```
eclass( cForm ).
eattribute( cForm, aTarget ).
is_a( aTarget, cString ).
changeable( aTarget ).
lower_bound( aTarget, 1 ).
ereference( cForm, rFields ).
containment( cForm, rFields ).
is_a( rFields, cAuto-fields-service ).
changeable( rFields ).
lower_bound( rFields, 0 ).
eclass( cAuto-fields-service ).
eattribute( cAuto-fields-service, aService-name ).
is_a( aService-name, cString ).
changeable( aService-name ).
lower_bound( aService-name, 1 ).
instance( obj2 ).
is_a( obj2, cAuto-fields-service ).
attrvalue( obj2, aService-name, 'updateSurvey' ).
instance( obj1 ).
is_a( obj1, cForm ).
attrvalue( obj1, aTarget, 'updateSurvey' ).
refvalue( obj1, rFields, [obj2] ).
```

Figure 5.8: *Mapping EMF to Prolog with SmartEMF (from [HCW07])*

5.3 Models and Ontologies

In literature, especially the modeling of information systems was subject to discussions. Many authors discussed the potential of ontologies to improve semantic formality in conceptual modeling for information systems specified with prominent modeling languages, such as UML [EW01, GHW02a, OHS02]. To this end, several publications arose that consulted the prominent ontology for information systems Bunge-Wand-Wevber (BWW) [WW88], a descendant of the earlier work of Mario Bunge [EW01, OHS02].

In [EW01], Evermann and Wand propose the enrichment of UML with ontological concepts by establishing a mapping between both. This shall foster the use of UML for conceptual modeling. Through the established mapping, the authors arrive at well defined semantics for UML models, which facilitates to giude the construction of conceptual models through intra- and inter-diagram integrity rules. The foundational ontology that underlies the ideas of the authors is the Bunge-Wand-Weber (BWW) ontology.

In [GHW02b], Guizzardi et al. advocate the use of ontologies for modeling languages. In detail, they analyze the *General Ontological Language (GOL)* [DHHS01] (see Section 6.1.1.3), developed and maintained at University of Leipzig, and define a mapping to different UML constructs. This reveals a number of relations between relevant ontology concepts and particular modeling constructs, which are also important for our work (see Section 6.1.2.1). In summary, the authors tried to achieve with GOL what Evermann and Wand aimed at with BWW. They explicitly mentioned this work and criticized serveral shortcomings of BWW that were more appropriately adressed with GOL.

In [BH03], Burrek and Herre propose the annotation of UML models with corresponding upper-level ontologies. The corresponding annotation information is stored in so-called *Ontological Extension of UML (OX-UML)*. Like other researchers, as a result the authors expect improvements in modeling with UML through gained semantic clarity and comprehensive axioms derived from the upper-level ontology. Furthermore, they aim at extensibility of developed extensions and interoperability with alternative representations, such as database schemas for software engineering.

Against the general trend to advocate conceptual models with ontological foundations in order to reach generally accepted benefits, such as sophisticated formal semantics or advanced integrity ensurance, Wyssusek et al. questioned the very idea of the whole approach [WK05]. The authors provided an analysis of the work of Wand and Weber, who created the BWW ontology based on the peceedingly developed foundational ontology for information systems by Mario Bunge. In this analysis they claimed that the development of BWW did not follow a scientific approach and concluded very negatively that "the project of ontology-based conceptual modeling appears to be impossible in principle" [WK05, p. 9]. Wyssuseks paper was answered with other publications of Guarino et al. and Opdahl [Opd06, GG06], who argued *for* an ontology-driven approach of conceptual modeling. The authors did not disprove Wyssuseks arguments but discussed them with proper examples and exposed that "there are no references to philosophical theories in ontology outside Mario Bunge's work" [GG06, p. 2], proving that Wyssuseks considerations were rather superficial.

In summary, it can be stated that the employment of ontologies in information system engineering is a well discussed topic, which was also used as proof of concept in the evolution of foundational upper ontologies (cf. Section 6.1.1.3). However, related work mainly focused on conceptual modeling, while modeling on a lower level of abstraction was not considered. Nevertheless, with the comprehensive related work in representing modeling constructs with semantic concepts and the discussions around emerging problems, we find a useful foundation to derive our own upper ontology, as documented in Section 6.1.

Chapter 6

Results

In the previous chapters, we explained the HybridMDSD approach which was designed to overcome the problems modern software industries suffer from. Moreover, we analyzed related approaches that already try to do so. In this chapter, we present the results of our work. To this end, we explain our efforts to implement HybridMDSD. Our presentation is subdivided into the three major parts of our approach covering its semantic core (in Section 6.1), its code composition framework (in Section 6.2), and its development process (in Section 6.3). Each of these parts gained a lot of attention in dedicated master theses [Brä07b, Her08, Zab08] and several publications [Loc07, BL07, PCG$^+$08, BL08, LG08b, LG08a, LH09].

6.1 Semantic Core

The semantic core of HybridMDSD is the central building block of our approach. Only with the enabling semantic structures between DSLs and DSMs, we are able to gain advantages for multi-DSL development, including consistency ensurance, integration code generation, change adaptation, and guidance (cf. Section 4.4). In this section, we present the details of this semantic core, whose main component is our ontology for software models, the *Unified Software Modeling Ontology (USMO)*. To this end, we explain our procedure for the design of USMO (see Section 6.1.1) and shed light on central concepts, relationships, and its axiomatization (see Section 6.1.2.1). To illustrate the application of the semantic core, we give a number of selected design examples in Section 6.1.2.2.

6.1.1 Preliminaries

This section gives a thorough overview of the design process for our upper ontology, USMO, which constitutes the semantic core of HybridMDSD. As depicted in Figure 6.1, we approached the challenge from three different angles. First, we established key principles for ontology design and selected an initial pragmatic ontology candidate, the ABC ontology (see next section). Second, we constituted a classification scheme for domain-specific languages, to identify most important DSL types and ensure an optimal conceptual covering in our ontology (see Section 6.1.1.2). Last, we comprehensively analyzed existing upper ontologies in the related context of information system engineering, to identify most common design patterns (see Section 6.1.1.3).

In Section 6.1.1.4, we discuss the result of this approach and clarify which choice we made in ontological design, central concepts, and relationships for our upper ontology.

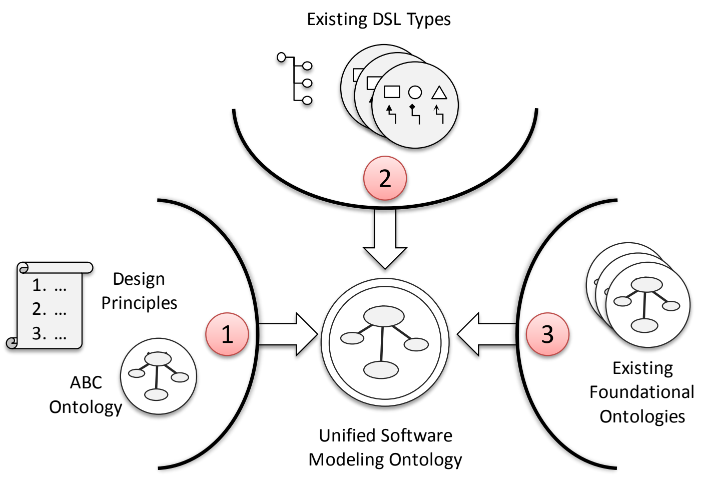


Figure 6.1: *Our approach to design an upper ontology for software models*

6.1.1.1 Design Principles

As a first step to narrow an appropriate semantic foundation for DSL integration, it was necessary to agree on crucial requirements for such a foundation. The HybridMDSD approach aims at a combination of the domains of software engineering and knowledge engineering, using an upper ontology. Leveraged reasoning and knowledge deduction capabilities of Semantic Web technologies enable us to provide advanced support for software developers. Yet, to establish semantic structures between domain-specific languages, at least a minimal understanding of therefore necessary semantic technologies is required. These, in turn, can not be assumed from software developers, which has major effects on the design principles for the central upper ontology.

A proper upper ontology that shall formalize the semantics of software models, on the one hand, needs to fulfill the requirements for sophisticated logics processing support. On the other hand, such an ontology must be straightforward enough to be usable in a software-intensive context. Thus, it is of vital importance that software developers do not need to struggle with ontological details but rather are able to transparently integrate such "foreign" development issues and can primarily focus on its advantages. To make the grade we declare three key principles that guide us through the design process of our upper ontology:

1. **Ease of Use:** The concepts and relationships of the upper ontology should be as familiar as possible with constructs known from software programming and modeling, respectively. This concerns both, the granularity of ontological entities and their naming. The naming should, in this respect, be independent from existing metamodel specifications but nevertheless precise and expressive enough to be easily understood.

2. **Limited Set of Concepts:** HybridMDSD is an enabling technology and prerequisite for integrated multi-DSL development. Hence, employed languages must be mapped to the central upper ontology *before* the actual modeling process starts (cf. Section 4.2). A limited set of available ontology concepts for language mapping is crucial to not overwhelm developers with the integration task before the actual work has begun.

3. **Limited Reach:** As already stated in previously demanded requirements, the integrating semantic structures should be "limited to information that is necessary for language integration" (see Requirement 7 in Section 3.4.6). It is, hence, not intended to declare the semantics of an entire software system, including detailed facets for various domain-specific

areas. This is rather implemented by particular DSLs. The primary purpose of USMO is to establish a *foundation for inter-model references*.

As additional corner stone and starting point for further analysis, we identified an ontology that seemed to be a promising initial candidate for our purposes.

An Initial Candidate: The ABC Ontology

One light-weight ontology which, because of its simplistic modeling approach, qualified as an ideal candidate for our purposes is the *ABC ontology* [LH01, HLL02]. The ABC ontology was designed in context of the *Harmony* project [Har02], a joint venture between the Distributed Systems Technology Centre, Australia, the Cornell Digital Library Research Group, USA, and the Institute for Learning and Research Technology, UK. The major goal of the project was to facilitate a better description of increasingly distributed digital library information. As an outcome, the core ABC ontology should enable for the integration and exchange of such information, by providing a well-defined basis for different domain-specific vocabularies [DHL03]. To this end, the predominant focus lies on "the ability to model the creation, evolution, and transition of objects over time" [LH01, p. 2].

Figure 6.2 depicts the relevant subset we choose to incorporate into this work. Please note that we have left out several concepts that had dedicated relevance for the domain of library content, but were superfluous for our purposes.

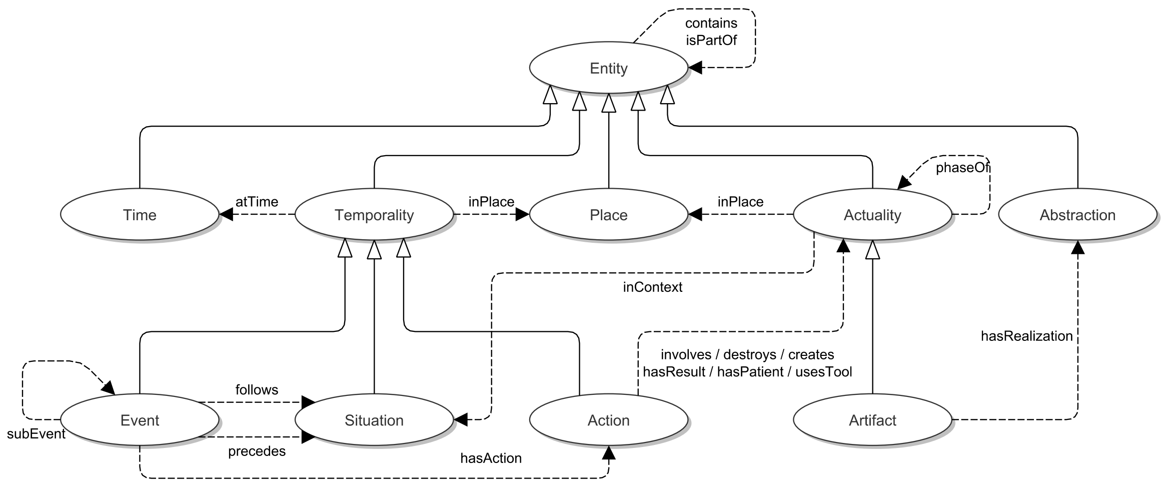


Figure 6.2: *A relevant subset of the ABC ontology (based on [LH01])*

The ontology is based on the top-level type Entity, which defines a containment relation for all other entities. Two other major top-level concepts are Temporality and Actuality, where the former explicitly models time, the latter represents tangible, sensible things. In our context of software modeling, we comprehend Actualitys as structural entities (e.g, data structures, files, or components), which persist in computer memory or permanent storage devices. Actualitys are realizations of Abstractions, a concept to express intangible concepts or ideas. An Actuality can have one universal or time-independent facet and many existential or time-dependent facets [Hun03]. In this regard, The existential facets represent different phases during its lifetime. To this end, each phase is inContext of a particular Situation. Changes between Situations are triggered by Events.

Although we knew that this ontology would most probably not suffice our requirements to describe the semantics of software models — already because it was created for an entirely different purpose — the ABC ontology provided an exemplary pragmatic ontology design that contained a number of important concepts for the description of real world behavior. Moreover, first practical experiments with the ontology proceeded successfully [Loc07] and confirmed our confidence in the chosen conceptual foundation. Thus, we decided to align and compare further efforts in designing a comprehensive upper ontology for software models with the foundational design of ABC.

6.1.1.2 Classification of Domain-Specific Languages

The three key design principles as well as the ABC ontology as initial candidate (cf. section above) provided a proper starting point for further efforts towards the development of a central ontology that should facilitate the DSL integration task. Now, to decide which top-level concepts and relationships are crucial to represent the semantics of common domain-specific languages, it was necessary to elucidate which kinds of languages exist and how they are characterized. To this end, we analyzed a number of domain-specific lanuages and existing approaches to language classification in related work [Spi00, MHS05][GS04, pp. 263], in order to design a classification scheme that is general enough to survey most important language kinds. Figure 6.3 shows the resulting scheme, originally introduced in [Brä07b].

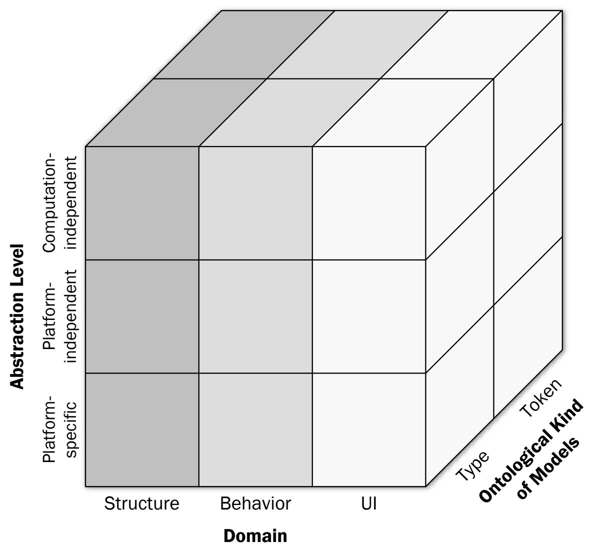


Figure 6.3: *Classification scheme for domain-specific languages (from [Brä07b]*

The cubic scheme is made of three dimensions, where each cell summarizes DSLs from the same *domain*, equal *abstraction level*, and matching *ontological kind*. In the following, we explain the actual meaning of the different dimensions.

Domain refers to the horizontal, technical domain a DSL is designed for, as introduced in Section 2.2.2. Concerning the chosen sub-categories, we mainly follow a slighlty altered version of the classification of Greenfield and Short [GS04, pp. 263].

Abstraction level refers to the degree of generalization, from a technical point of view. Obviously, we follow the layered architecture of the OMG MDA specification, introduced in Section 2.1.3.2. The explicit consideration of the language abstraction level, is mainly motivated by the intent to integrate DSLs of different abstraction levels.

Ontological kind of models refers to two fundamentally distinct types of models, first identified by Kühne [Küh06]. The first type denotes *type* models which are probably the most common to software developers. Type models *classify* objects based on their properties and behavior. Appropriate alternative terms for type models are *schema model*, *classification* *model*, *universal model*, or *intensional model*. Prominent examples are UML class diagrams. In contrast, *token* models do not classify or abstract from objects, they rather act as direct representations. In UML, typical examples are the object diagram and the sequence diagram. Proper alternative terms are *instance model*, *singular model*, or *extensional model*.

For the given domains, various language types and examples can be enumerated.

Structure Modeling Languages

For example, *structure* modeling languages comprise DSLs on CIM, PIM and PSM level, and both, type and token models. This includes *domain languages*, *type languages*, *object languages*, and *deployment languages*.

Domain languages are conceptual modeling languages that are usually employed in the early development phases of software projects, such as requirement engineering or domain analysis. Corresponding languages represent real world concepts and relationships and, thus, are usually *computation independent* [Myl92]. Typical examples are UML analysis class diagrams or *Entity Relationship (ER)* diagrams [Che76]. Domain languages usually yield *type models*.

Type languages count to those kinds of DSLs that are probably most familiar to software developers, because they are the common means to define data structures in software engineering. Prominent examples are UML design class diagrams, SysML block definition diagrams [Sys07, pp. 35-40], or Entity Relationship diagrams. But also DSLs to model more complex composite types or colaborations between such types, like the *Fundamental Modeling Concepts (FMC)* notation to model compositional system structures [TAW03], can be considered as type languages. Type languages result in *type models* and are, due to their technical nature, *computation dependent*.

Object languages specific valid or invalid runtime states based on object configurations, consisting of a set of attributes and corresponding value allocations. Corresponding *token models* are usually used to illustrate how a system evolves over time. Examples are UML object diagrams or the *Common Warehouse Metamodel (CWM)* instance metamodel [CWM03, p. (4-55)].

Deployment languages are used to specify, visualize, and simulate the assignment of software assets (or artifacts) to physical runtime components (or nodes). Beside generic approaches, such as UML deployment diagrams, also domain-specific solutions can be found, like *AUTODeploy* [WSNW07] proposed by Jules White. Because deployment language models represent actual runtime configurations, they constitute *token models*. Like object language instances, such models are either *platform-independent* or *platform-specific*.

Behavior Modeling Languages

Modeling behavior is admittedly one of the most difficult challenges the MDSD community must face. Moreover, it is commonly questioned whether an abstract specification of usually very fine-grained descriptions of entity reactions in dependence of system events, contextual information, and adjacent entities, can be formalized other than with imperative code structures [SK06]. However, several approaches to express system behavior exist. This includes *statechart languages*, *workflow languages*, and *collaboration languages*. Altogether, such languages usually yield *token models*. A dedicated discussion around this topic is given in 6.1.2.1.

Statechart languages facilitate the specification the entity behavior based on particular states. To this end, a state is a situation where several *invariants* hold true for an entity and, thus, is defined through a set of concrete attribute assignments for that entity. Statecharts were originally introduced by Harel [Har87] and later incorporated into the UML specification as state machines [UML07, p. 519]. As statechart languages define the behavior of software entities, they are to be classified as *computational dependent.*

Workflow languages allow to specify workflows, which are, according to a widely-cited article, "a collection of tasks organized to accomplish some business process" [GHS95]. In context of information system modeling, workflows are usually considered as "computerized [...] automation of business process" [Hol95, p. 6]. Corresponding languages combine control-flow elements (e.g., forks and joins), data-flow elements (e.g., objects and documents), and operational elements (e.g., tasks and actions), and are usually shipped with appropriate workflow systems that manage workflow execution. An important standard for CIM-level workflows is the OMG's *Business Process Modeling Notation (BPMN)* Specification [BPM06]. Prominent examples for platform-independent and platform-specific workflow languages are the *XML Process Definition Language (XPDL)* [XPD05] and the *Web Services Business Process Execution Language (commonly called BPEL)* [BPE07], respectively.

Collaboration languages are used to specify behavior that arises from cooperated actions between system entities. To this end, UML provides a number of languages, from CIM-level use case diagrams down to computation dependent sequence diagrams. Another interesting example for a collaboration language are Fujaba story diagrams, developed with the Fujaba Tool Suite at the University of Paderborn, Germany [FTSDT09]. Story diagrams incorporate different parts of UML, such as class diagrams, activity diagrams, and collaboration diagrams, and facilitate the specification of imperative object-oriented programming code with graph-based abstract models [FNTZ98].

User Interface Modeling Languages

The efforts around the model-driven specification of user interfaces in software engineering are commonly summarized under the umbrella of *Model-Based User Interface Development (MB-UID).* Corresponding modeling languages aim at an *abstract* description of UI ingredients, such as user interaction, navigation, and visual representation. To this date, such abstract specifications could not prevail. Instead, a number of low level *WYSIWYG (What You See Is What You Get)* modeling tools emerged, which allow for easy creation of *concrete* user interface presentations [Mol03]. Recent examples of such tools, such as the Flash-based UI technology *Adobe FLEX* [Inc09] or the competitive approach of Microsoft called Silverlight [Cor09b], witness this trend. However, several approaches to abstract specification exist and are increasingly important through the accelerated development of multimodal and ubiquitous applications. We divided corresponding languages into *task languages*, *dialog languages*, *presentation languages*, and *interaction languages.* As UI languages usually yield direct representations of interaction, dialogs, and widgets, they uniformly denote token models.

Task languages result in task models, which define how a user can reach a goal in an application domain. In context of software engineering, such a goal is usually to query a system or to modify its state [PMM97]. To this end, task models usually allow to create hierarchical task structures which break down higher-level user tasks into lower-level application tasks, as proposed with the *ConcurTaskTrees (CTT)* notation [PMM97, MPS02]. Since task languages allow to represent a structured way to achieve particular goals but do not restrict how corresponding tasks need to be implemented, such languages are to be classified on CIM-level.

Dialog languages allow to specify navigation paths between user interface dialogs. To this end, they provide dedicated modeling means, such as events, transitions, start, and end nodes. A dialog model can be considered as "an intermediate model between the task model and presentation model" [Luy04, p. 23]. It is more precise than task models since it defines a sequence of dialogs to perform a task and less detailed than presentation models, which define the visual appearance of these dialogs. Hence, dialog languages are never computational independent. An industrial example for a dialog language is the jPDL pageflow language for JBoss Seam applications (cf. Section 5.1.3.1).

Presentation languages are used to specify the actual visual representation of an application's user interface. A basic distinction can be made between *abstract*, platform-independent and *concrete*, platform-specific user interface descriptions [LVM$^+$04]. *Abstract UI (AUI)* models define interaction spaces based on higher level criteria, like task descriptions, independent of any modality of interaction. *Concrete UI (CUI)* models contain explicitly detailed information about margins, colors, and the alignment of widgets along a hierarchic structure. How model-driven approaches can realize platform-independent UIs from loose goal descriptions, over task, AUI, and final CUI models, was presented in the pan-European research project EMODE [Con07b]. Appropriate examples for low-level UI languages abound. Besides declarative formats to specify usually imperatively created UIs, such as CookSWT for Java Swing and SWT [Yua07], entirely declarative language examples are the *XML User Interface Language (XUL)* [(MD09], XAML [Nay07], or the above mentioned Adobe FLEX.

Interaction languages facilitate the specification of UI reactions to particular user input. Although no explicit languages can be found in literature, respective tool suites partially provide domain-specific solutions for this kind of problem. For example, JBoss Seam provides dedicated JSF tags to bind particular user input fieds to predefined Hibernate validation routines [KMR$^+$08, pp. 207-214]. In case entered data does not adhere to corresponding validation patterns, the UI responds with red shaded widget backgrounds or appropriate icons to indicate the error.

For a more detailed elaboration of DSL classifications and exemplary languages, the reader is kindly referred to [Brä07b, pp. 50-60].

6.1.1.3 Review of Upper Ontologies

The employment of ontologies in software engineering is no new idea, as already shown in Section 5.2.3. Thus, quite a number of ontologies were developed especially for conceptual modeling in the area of information system engineering [Gua98]. Therefore, we performed a comprehensive analysis of existing upper (or foundational) ontologies in this area. The major goal of this analysis was the identification of prominent recurring elements and design patterns to establish necessary expertise for the design of design of our own ontology.

Bunge-Wand-Weber

We start our review with the probably most prominent *Bunge-Wand-Weber (BWW)* ontology, which is, at the same time, the oldest approach to conceptual modeling, firstly published in 1988 [WW88]. The ontology is originally based on Bunge's "treatise on basic philosophy" [Bun77, Bun79]. To this date, well over 100 publications document the influence of BWW in different areas of conceptual modeling, such as ontological evaluation of modeling grammars or information systems interoperability [WK05]. Besides various affirmative applications of BWW, the ontology was also controversially discussed [Wys06, GG06, Opd06], which proves its maturity. Figure 6.4 depicts an excerpt of selected main concepts of BWW. Comprehensive descriptions of the ontology can be found in [WW95, OHS02, EW05].

The central concept in BWW is Thing, which follows the Aristotelian idea of *substance*, i.e., a physical being that doesn't change identity over time [GHW02a]. Things possess Propertys,

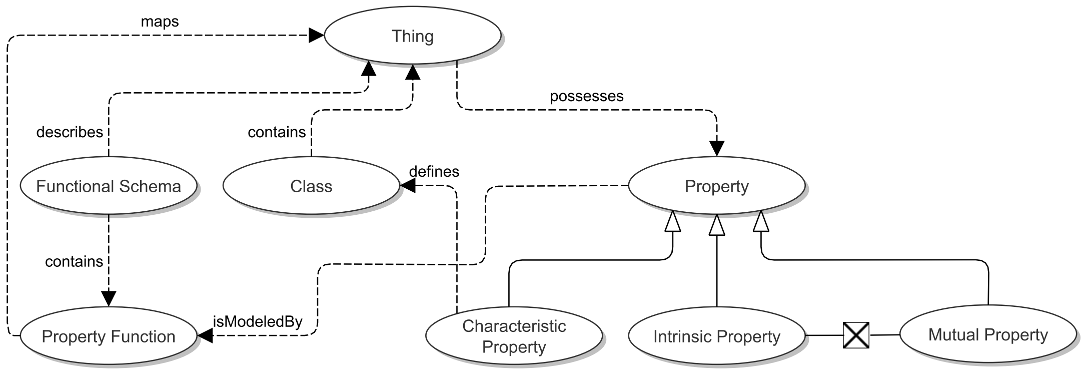


Figure 6.4: *Selected structural concepts of the Bunge-Wand-Weber ontology*

which represent, in contrast to the common terminology of software modeling languages, concrete manifestations of features. Thus, the property “name” of one entity is not the same as the property “name” of another entity, even if they have the same value. Property values are calculated by Property Functions that map a Thing to a value in a so-called “property codomain”. To this end, BWW distinguishes between Intrinsic Propertys and Mutual Propertys, where the former denotes inherent and the latter relational properties. A Functional Schema aggregates several Property Functions and, thus, provides the *intensional* definition for Things. In contrast, a Class does not specify Things intensionally, but rather bundles Things that exhibit particular properties. These Class properties are considered as properties “in general” and called Characteristic Propertys [OHS02]. Therefore, a Class represents the *extensional* definition of Things.

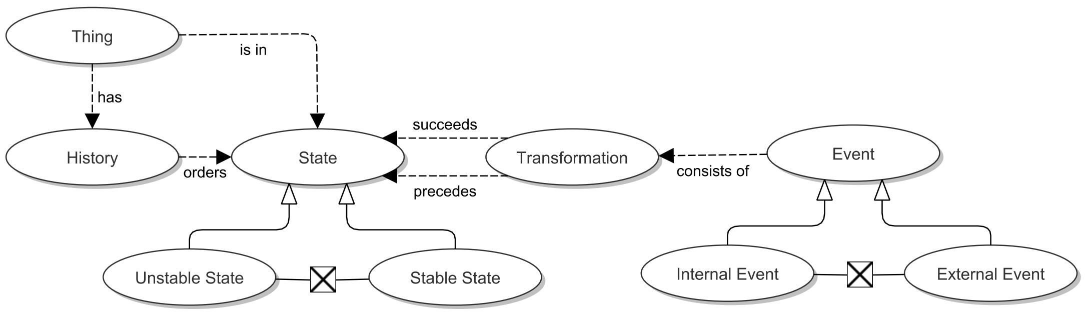


Figure 6.5: *Selected behavioral concepts of the Bunge-Wand-Weber ontology*

Concerning behavioral modeling consider Figure 6.5. BWW provides a very important and precise definition of states, whereafter a State is a “vector of values for all property functions of a thing” [WW95]. Another contribution, and also an improvement from the perspective of the ABC ontology, is that state changes are performed in Transformations, when an Event occurs, rather than by events directly like in ABC. This aligns nicely to statechart and workflow languages, which often employ dedicated constructs for transitions. In contrast, the distinction between stable and unstable states as well as external and internal events seems to be superfluous in our context. In a modeled software application, such concepts clearly refer to domain-specific, time-critical parts of the program, which should be specified in dedicated DSLs rather than captured in an integrating upper ontology.

GOL / GFO

The *General Ontological Language (GOL)* is a formal framework for ontology development, delivered by the OntoMed project of the University of Leipzig, Germany [Ont07]. GOL is comprised of a system of top-level ontologies which is called *GFO (General Formal Ontology)* and contains major concepts and axiomatization for GOL [HH03a]. An excerpt of the GOL/GFO's major concepts is depicted in Figure 6.6.

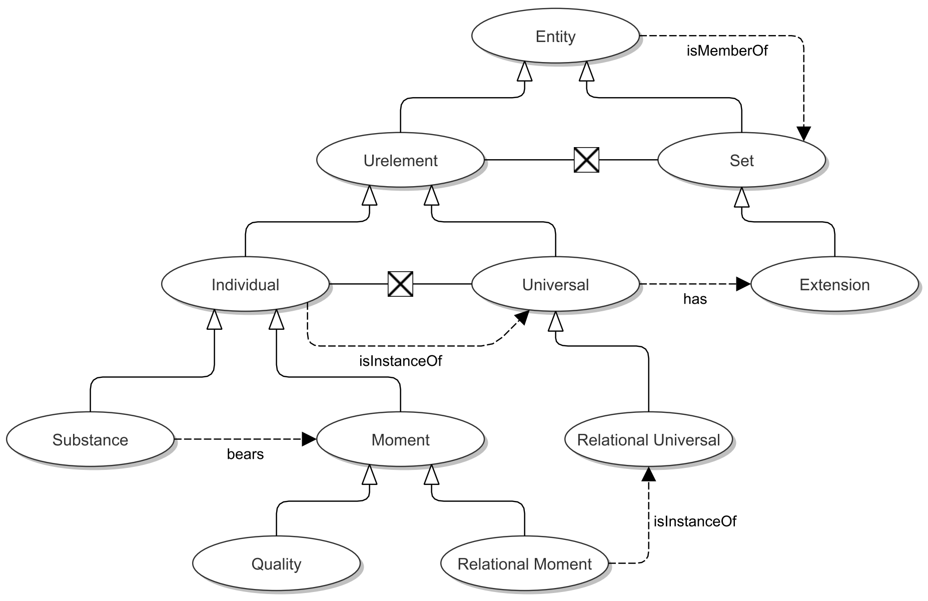


Figure 6.6: *An excerpt of GOL/GFO basic concepts (based on [GHW02a])*

In principle, besides the two top-level concepts Urelement and Set and some dedicated notions for space and time, GOL/GFO shares a lot of concepts with BWW. Substance corresponds to Thing, Functional Schema corresponds to Universal, and Property corresponds to Moment, including so their sub concepts, where a Quality denotes an intrinsic feature and, therefore, corresponds to Intrinsic Property. Hence, Relational Property corresponds to BWW's Mutual Property accordingly.

In essence, the main contribution of GOL/GFO is an explicitly named distinction between Universals, Individuals, and Sets, which the authors argue every upper ontology should be based on. Thus, they reject the *pure* set-theoretic approach, in that universals are only *sets of individuals*, i.e., their extension. The authors argue that this would "not do justice to the intensional character of universals" [DHHS01]. Instead, they propose to incorporate both, the extensional and the intensional ontology of universals and explicitly define them as "patterns of features which are realized by their instances" [DHHS01].

After its introduction, GOL/GFO evolved further and grew significantly. This led to an initially missing explicit concept for temporal entities, the Occurrent [HH03a]. In this regard, Heller and Herre comprehensively defined how to model changes of individuals over time. To this end, they introduced the notion of *abstract substance*, which represents an individual over a certain range of time. Assume, for instance, a Substance (an individual) that exists at one time and a Substance that exists at another time, both with different properties. Then, these Substances belong together (they are *ontically connected*) if they belong to the same "base" Substance covering both points in time, the abstract substance. A more illustrative example is a person that looks different in its infancy than when it is an adult.

DOLCE

The *Descriptive Ontology for Linguistic and Cognitive Engineering (DOLCE)* was developed as a modul for the Formal Ontologies Library in context of the WonderWeb project [MBG$^+$03, Won04]. The purpose of DOLCE, which is to describe real world entities primarily represented in natural language, actually differs from our context. Nevertheless, we include the ontology into our review of foundational ontologies because it (1) is another, comprehensive research approach to conceptual modeling and (2) had major influence on the (more relevant) UFO ontology, introduced below.

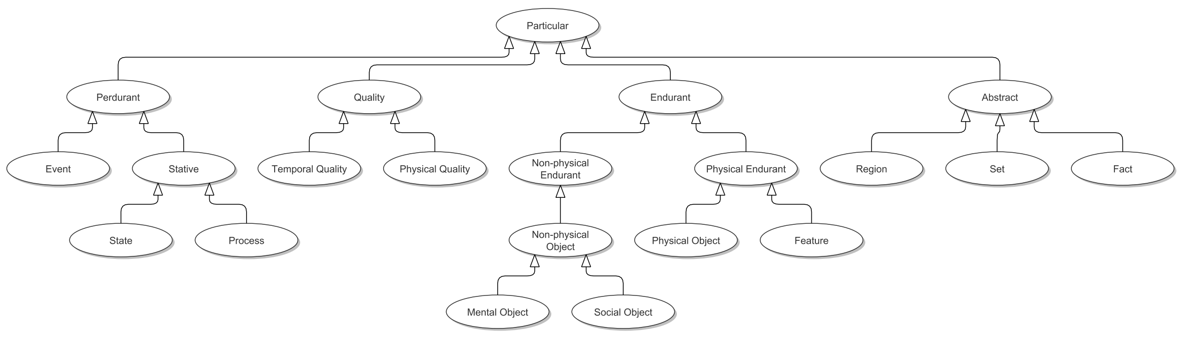


Figure 6.7: *An excerpt of major concepts of the DOLCE ontology (based on [MBG+03, p. 14])*

DOLCE predominantly distinguishes between *enduring* and *perduring* entities. Although we already gave several examples for these kinds of concepts, we did not yet properly clarify on this basic distinction, which is done here. Enduring entities (or *continuants*) are wholly present in time, i.e., all their proper parts are present at any time they are present. Perduring entities (or *occurrents*), instead, are only partially present in time. Philosophers say that endurants *are in time* while perdurants *happen in time* [MBG+03, p. 15]. In previously introduced ontologies, concepts for modeling of time-related entities, such as the BWW Event or the GOL/GFO Occurrent, are perdurants.

The strict differentiation between Endurants and Perdurants reaches down to the formalization of properties. To this end, it is defined that an Endurant can "genuinely" change over time, i.e., it may *inhere* incompatible properties at different times. For example, a piece of paper can have the two incompatible properties "is white" and "is yellow", at different times. To this end, DOLCE properties are always equiped with a particular time value. Thus, Endurants are not considerd as entities that are *wholly* present in a three dimensional space as long as they exist, but rather as "space-time worms" [MBG+03, p. 10].

Moreover, when considering DOLCE's top-level concepts, it reveals that the ontology does not specify a single concept for types, i.e., universals. Even properties are modeled as Qualitys, i.e., unsharable individuals whos values are defined as locations in the conceptual space of the quality type and represent, thus, concrete property instances. Thus, "DOLCE is an ontology of particulars" [GGM+02], which is proved by the fact that every concept is derived from the root concept Particular.

UFO

The *Unified Foundational Ontology (UFO)* is an approach to unify the existing, most comprehensive foundational ontologies BWW, GOL/GFO, and DOLCE, and is, thus, probably the most thoroughly researched foundational ontology for conceptual modeling to date. The central goal of the UFO developers was to create a synthesis of these ontologies that is tailored to conceptual modeling, and overcomes their "sever limitations" [GW05]. In detail they demand, for instance, the lack of agentative modeling in BWW and GOL/GFO, and a proper distinction between entities and sets.

One of the major contributions of UFO is its comprehensive ontology of universals. It devides universals into Mixin Types and Sortal Types and the latter, in turn, into Base Type, Role Types, and Phase Types. The common denominator for "common" types is Sortal Type, which introduces

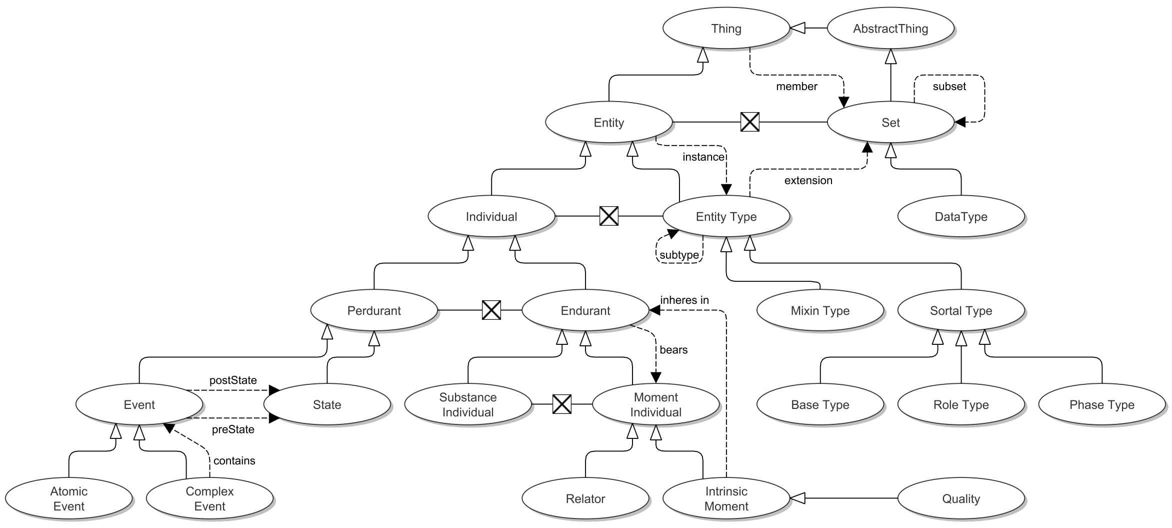


Figure 6.8: *Main concepts of UFO (adapted from [GW05])*

among others the criterion of identity for its instances to facilitate the judgement whether two instances are the same. Base Types are *rigid* universals, i.e., types whose instances are necessarily and also remain their instances (e.g., Montain or Person). Phase Types, instead, are *anti-rigid* types whose instances may or may not be instances of it. To this end, a Phase Type subclasses both, a particular Base Type and Phase Type. For example, Town is a phase subtype of the Base Type City. Role Types are anti-rigid Sortal Types, which do not subtype particular Base Types but are rather related to them. For instance, a role sub type for City is Destination City. Mixin Types are non-sortal and, hence, can be shared between particular disjoint Sortal Types. An example is the Mixin Type Customer, which could be applied to the sortal type Person (and gives the “consumer”) and to the sortal type Company (which gives the “corporate customer”). For further explanations around rigidity and sortality, the reader is kindly referred to [GWGvS04] and [GW05].

6.1.1.4 Discussion

The HybridMDSD architecture necessitates a sophisticated semantic core, which, on the one hand, needs to be comprehensive and well axiomatized to facilitate optimal assistance during DSL and model integration. On the other hand, this core must be lightweight and limited enough to not overstrain software developers with the necessary tasks preliminary to language integration (cf. Section 6.1.1.1). The ABC ontology was one initial candidate, which could convince in these repects at a first glance [Loc07]. However, practically applied modeling languages and theirein encapsulated semantics are difficult to describe and ABC is only vaguely axiomatized. In contrast, existing foundational ontologies for conceptual modeling provide huge concept/relationship sets and rich axiomatization, which might be better suited in our context. Unfortunately, these ontologies are very complex and difficult to communicate. In the following, we discuss the suitability of ABC and existing foundational ontologies for our purposes.

Suitability of the ABC Ontology

The ABC ontology, as presented in [LH01, Hun03] and [DHL03], provides only weak axiomatization. For example, the notion of Abstraction is loosely described as a concept to describe “entities that are pure information or concepts” [LH01]. The fact that an Abstraction “stands in contrast to the Actuality category” and both are connected with the hasRealization relation, allows to conclude that Abstractions might represent the *intension* of tangible things. Anyhow, following the loose axiomatization, allows an Abstraction to have arbitrary Actualitys as realizations, which is inconsistent with object-oriented software systems. There, an object necessarily instantiates its defining class and remains instance of this class during its entire lifetime. More complete approaches, such as the BWW ontology, clearly emphasize that a Functional Schema acts as *template* for particular Things.

Another shortcoming of ABC is the lack of proper top-level concepts for the vital distinction between universals, particulars, and sets. An appropriate concept for sets, for instance, is simply missing in the ontology. Yet, the analysis of existing DSLs showed several examples for type and token languages. To properly represent such a basic distinction, a dedicated specification of the relationships between types and instances in both ways, intensionally and extensionally, is crucial. This is not provided by the ABC ontology.

Moreover, the concept to model changes between States directly via bridging Events is insufficient for the integration of corresponding behavioral languages. As already highlighted in Section 6.1.1.2, various DSLs exist that explicitly define modeling constructs for state transitions. To ensure the seamless integration of such DSLs, it is important to facilitate the specification of equality between selected constructs. To this end, the commonly available transition construct needs to be backed with an appropriate ontological concept.

An argument in favour of ABC is the basic distinction between enduring and perduring entities. The top-level concepts Actuality and Temporality match perfectly well with differentiations of more comprehensive approaches and are basic prerequisites for representing entities of structural and behavioral modeling languages. In addition, the ABC approach to model the change of entities over time is reasonable and straightforward. Deviding an enduring entity into universal and existential facets to model different phases of its lifetime is far more easy to understand than the specification of four-dimensional, time-indexed properties as proposed by DOLCE. More sophisticated considerations following the same approach, such as in GOL/GFO, also justify this approach.

Suitability of Existing Upper Ontologies

Our review of related work revealed a number of foundational ontologies with vital conceptualizations that are obviously far more comprehensive than the one provided by the ABC ontology. However, one important detail needs to be taken into account: all of the encountered approaches exclusively focus on *conceptual modeling* scenarios. This is important to recognize, since it documents the lack of research done in the area of *computation-dependent* modeling, which we need to serve. So far, significant efforts have been invested to map particular upper ontologies to UML, *the* general purpose modeling language for software models [EW01, GHW02a, GHW02b, GWGvS04, EW05]. However, corresponding papers had a limited focus on conceptual modeling. Thus, Evermann and Wand even state that "derived [mapping] rules might not be applicable for system design, i.e. software modeling" [EW05]. Indeed, trying to describe user interface elements or data management namespaces with real world semantics, does not occur to be very sensible.

Another observation is that the analyzed upper ontologies seem to be far too complex to be practically applicable. GOL/GFO currently counts 77 concepts and 67 properties [GFO09] and also BWW, with its 47 concepts, contains too many concepts that are of little practical use for us (cf. Table 6.1). BWW defines, for example, the notion of Law, which enable to formalize property restrictions on Things. Such restrictions can be specified as conditions for States or Transformations of Things and, hence, denote their behavior. However, the envisaged upper ontology aims at *integration*. This means, *existing* models, which already specify a system, need to be related to each other based on a commonly applicable semantic foundation. In this respect, the value of behavior-deciding laws is questionable, because they shift the actual system specification from the modeling technological space to the ontology technological space. This needs to be avoided.

Recent examples show that the involvement of rich concept sets is not absolutely necessary to ground modeling languages on semantic foudations. Kurtev, for instance, employs only four different concepts for this purpose [Kur07]. He claims that too much complexity of ontological foundations questions its usability in a software modeling context and that additional theories can be added as needed. We think, however, that a tradeoff between complexity and such a minimal approach as Kurtev's will be adequate for our purposes. A sophisticated ontological foundation for the integration of *various* solution space modeling languages needs to cover the conceivable most important theories without the obligatory need for case-based extension.

	BWW	GOL / GFO	DOLCE	UFO
Background	Bunge's Treatise on Ontology	The Onto-Med project	Wonderweb Foundational Ontologies Library	Guizzardi's dissertation
Concepts	47	39+ (77 to date)	77	39
Application Areas	Information systems modeling, modeling language evaluation	Bioinformatics, modeling language evaluation	Ontology evaluation, semantic application server	Business modeling, analysis of structural conceptual models

Table 6.1: *Comparison of selected upper ontologies (adapted from [Brä07b])*

Conclusion

The preceding explanations have in pinciple shown that for the context of this work (1) the initially proposed ABC ontology is insufficient and (2) existing foundational ontologies are too complex and inappropriately scoped. Hence, we will define our own upper ontology, which aims at a partial description of domain-specific languages in order to facilitate language integration. To this end, it is important to adhere to well established ontology design patterns, which we already noticed in the reviewed upper ontologies. In the following, we enumerate identified recurring patterns and estimate their relevance for an upper ontology for software models in HybridMDSD.

Entities and Sets: The most fundamental top-level distinction is between entities and sets. The notion of sets could be recognized already in BWW, with the concepts Class, Kind, and Natural Kind. Later, with GOL/GFO, an explicit differentiation was presented that also influenced the most sophisticated upper ontology to date, UFO. As the employment of sets, which aggregate particular entities of certain types, is a prominent construct in software engineering, we consider corresponding concepts as vital.

Universals and Particulars: Of course, the differentiation between universals and particulars, i.e., types and instances, is another crucial feature in software modeling languages, as already proved with the availability of type and token models (see Section 6.1.1.2). The proper relationship between both concepts respecting both intensional and extensional meaning, is likewise important, as it perfectly integrates with the notion of above mentioned sets.

Endurants and Perdurants: These two ontologically different categories where predominantly explicitly named in DOLCE. But also the lightweight ABC ontology specified a dedicated distinction. Because both, structural and behavioral domain-specific languages, need to be represented and integrated, we explicitly demand both types of concepts in our upper ontology for software models.

Change Over Time: As already mentioned above, the modeling of the change of entities over time through a proper distinction between their universal and existential facets, like it is done in ABC and GOL/GFO, appears to be straightforward, logical, and easy to communicate. Instead, more powerful conceptualizations, such as proposed by DOLCE, which allow for time-indexed property specification and the definition of “space-time worms”, are rather difficult to understand.

Intrinsic and Relational Properties: With exception of DOLCE and ABC, the analyzed upper ontologies distinguish between properties that describe one single entity and properties that relate two or more entities. This distinction appears familiar for software models, especially in DSLs for data structure specification.

Time and Space: Both, GOL/GFO and DOLCE, exhibit complex theories for time and space. As already mentioned above, we anticipate that such theories are superfluous in our context, because software models are usually time and location-independent.It is rather important to facilitate the specification of time-relative locations between different entities. In case additional concepts are needed, appropriate core ontologies can add the missing conceptual framework.

6.1.2 Unified Software Modeling Ontology

In this section we introduce our upper ontology, the *Unified Software Modeling Ontology (USMO)*, which has been designed to facilitate multi-domain engineering within the HybridMDSD approach. Which approach we followed furing design, is explained in the preceding section sections. Here, we will start with a consideration of major concepts and relationships in Section 6.1.2.1. In Section 6.1.2.2, we will present particular design examples that shall clarify the way to use our ontology.

6.1.2.1 Concepts and Relationships

This section introduces the most important concepts and relationships of UMSO. The current version of USMO contains 27 concepts and 28 properties (56 incl. inverse properties). After the thourogh introduction to existing upper ontologies and therein recurring ontology design patterns in Section 6.1.1, the fundamental meaning of these entities should not necessitate further explanation. Thus, we will introduce important parts of our ontology mainly by using diagram presentations and informal textual descriptions. For further details, the reader is kindly referred to Appendix A.1 and A.2.

Foundational Concepts

On the topmost level, we devide the modeling world into Entitys and Sets, as depicted in Figure 6.9. All modeling entities in USMO are represented as either Universal or Particular *and* either Actuality or Temporality. The meaning of Universals and Particulars should be clear. Concerning the concepts Actuality or Temporality, we follow the terminology of the ABC ontology enduring and perduring entities. Hence, this top-level distinction matches perfectly with both, (1) our modeling language classification in Section 6.1.1.2 and (2) the major design patterns of upper ontologies in Section 6.1.1.4.

Concerning (1), we implement a proper separation of *type* and *token* models and simultaneously respect the distinction between structural and behavioral modeling languages (cf. Figure 6.3). Regarding, (2) we follow the differentiation between endurants and perdurants explicitly modeled in DOLCE. The distinction between Sets, Universals, and Particulars follows the consideration of ontological instantiation, made explicit by Herre et al. and later applied in UFO.

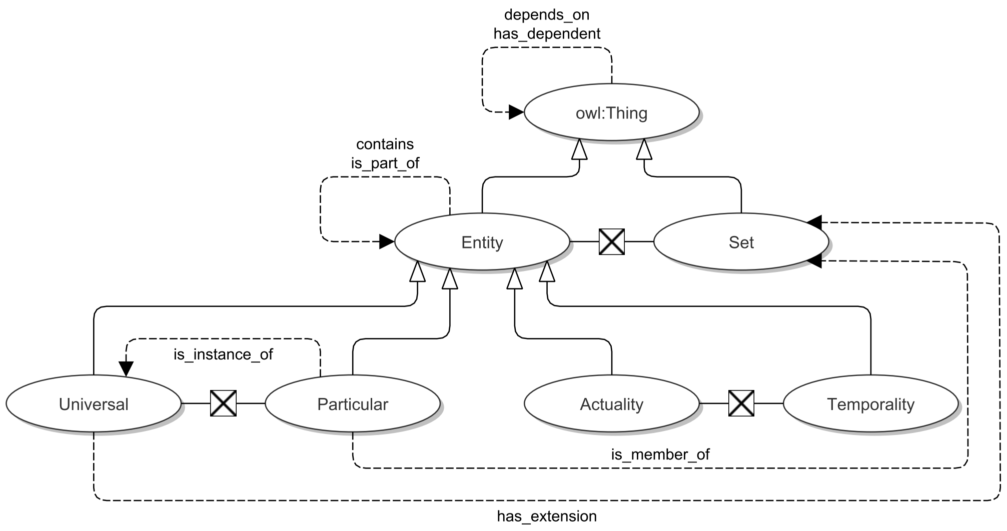


Figure 6.9: *USMO top level concepts and relationships*

This foundational framework for all other USMO concepts allows to represent structural and behavioral model elements on different levels of abstraction, which each of the identified modeling language types incorporates. Also user interface languages can be described using this approach, as explained below in Section 6.1.2.2. Moreover, this framework allows to specify *universal* temporal entities, a feature that is strikingly absent in existing upper ontologies. A distinction between universal and particular temporal entities is crucial in software modeling, as it allows to separate behavioral patterns from their actual execution. Recall the illustrative example from Section 4.3, where a *Service DSL* allowed to specify invokable data modification services. We consider such service *definitions* as Universal Temporalitys, which are instantiated through actual service calls, i.e., particular temporal Actions (cf. Section 6.1.2.2).

The most important relationship in USMO is the *existential dependency* relation depends_on, since it subsumes all other properties in our ontology. This way, all established connections between modeling entities, which are specified based on USMO in a corresponding knowledge-base, finally implement a complex dependency graph, which easily allows to yield consistency statements. Corresponding consistency rules need to be specified only once for the entire semantic core. Another core relationship is *containment* (called contains), which facilitates to model transitive, non-shareable ownership relations.

Representing Structure

Describing structure modeling constructs with USMO on a raised level of abstraction, i.e., on a classifying universal level, is mainly performed with the concepts Schema and Property, as depicted in Figure 6.10. The Schema concept denotes the intensional specification of structural individuals and is, therefore, an Actuality Universal. It, thus, corresponds to structural type definitions in software engineering and is defined through its properties. These, in turn, are modeled with the Property concept. More precisely, we distinguish between *intrinsic* and *relational* properties, common to the analyzed approaches. To this end, a Schema consists_of Intrinsic Propertys and a Relational Property relates Schemas.

On token level, the modeling of structural entities is implemented with the concept Substantial and, therefore, subsumed by Actuality and Particular (see Figure 6.11). Corresponding property individuals denote Qualitys and Relators. Substantials and Qualitys are connected through the

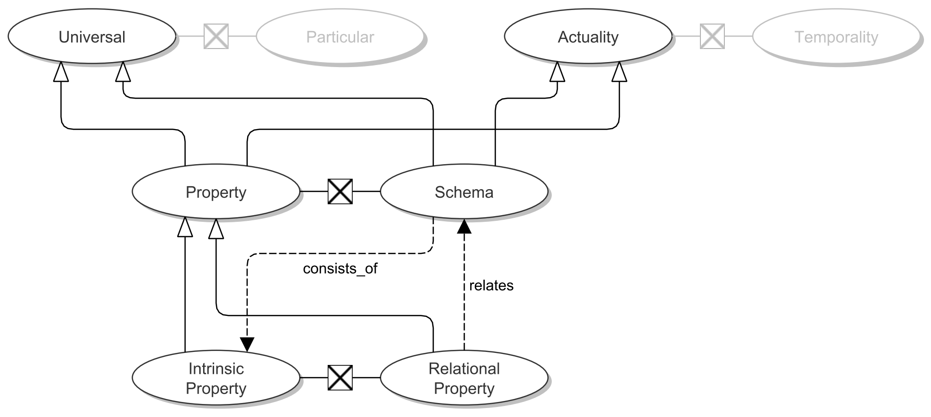


Figure 6.10: *Universal concepts to represent structure*

specialized containment relation bears, while two or more Substantials are related through the links relationship.

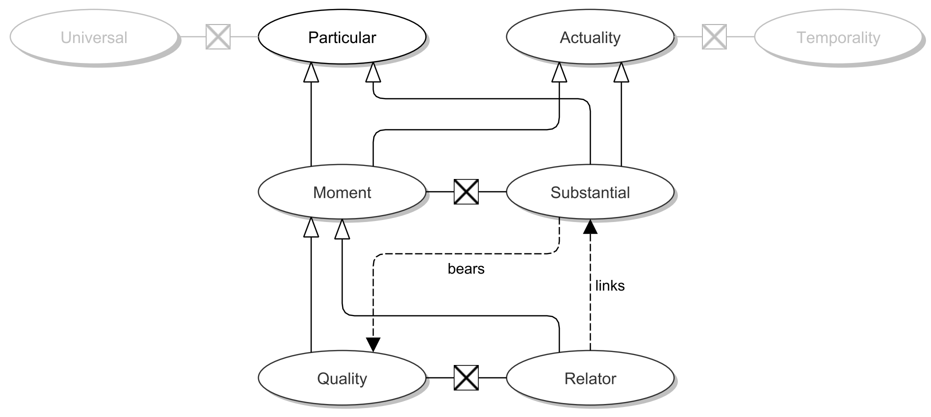


Figure 6.11: *Particular concepts to represent structure*

Representing Behavior

Behavior modeling with USMO is equally straightforward like structure modeling (cf. Figure 6.12). Although we realized that behavior models are usually token models, we identified scenarios where it is useful to *classify* particular temporal entities. It might, for instance, be necessary to represent dedicated types of events that occur in a software system, like the mouse click event in the user interface. For this purpose, we define the Signal concept. Another prominent example are service, operation, or method declarations. Here, the Behavior concept is used to represent action patterns, i.e., *templates* for particular system behavior.

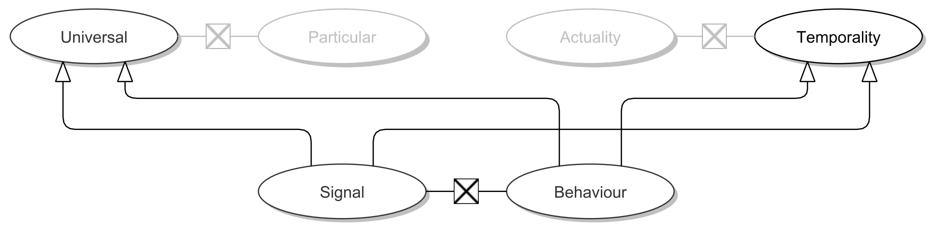


Figure 6.12: *Universal concepts to represent behavior*

Figure 6.13 depicts further concepts to model system behavior on the level of particulars. Thus, corresponding type defintions uniformly denote Particular Temporalitys. The concept Life, for instance, may be used to associate stateful behavior with related Particulars (see next section). To this end, we classify state-like entites with the root concept Situation and, thus, follow the structure of the ABC ontology. Situations represent system states in different granularity. Depending on the scope, they might denote the state of single objects, pages in a user interface wizard, or even the overall state of the surrounding system. Corresponding specializations are Activitys, which represent continuously executed system operations, and previously mentioned States, which denote a particular set of conditions for a dedicated set of properties (see below).

The reader might wonder why necessary concepts for the representation of stateful behavior are missing on the level of universals and, instead, appropriate concepts are defined as Particular Temporalitys. Now, the common perception seems to contradict with this approach. As example, take a UML statechart diagram, which is connected to a UML class definition. As the corresponding class correctly denotes the intensional type for its instances, it seemed to be natural to classify corresponding behavioral patterns, such as states, transitions, or actions, as Universal too. After all, statecharts specify the behavior for *all* the instances of corresponding class definitions.

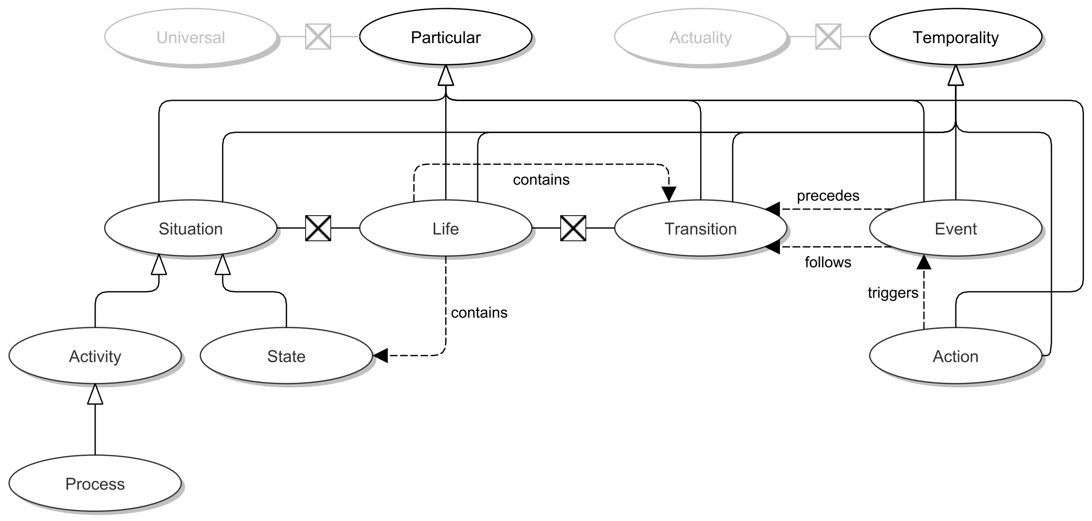


Figure 6.13: *Particular concepts to represent behavior*

However, as correctly introduced with the BWW ontology, a state denotes a particular situation in that certain invariant conditions hold. Hence, a state itself simply cannot be instantiated but rather must be "reached". If we would assume that statecharts and their parts would denote universals, then an instance of a class that is connected to such a statechart, had to be connected to an *instance* of the statechart as well. Now, recall that all states of the statechart define certain invariants that hold for a set of attributes. Thus, if two class instances would meet the invariants of the same state, both instances would be in the very same state instance, which would be inconsistent.

As a last remark, it is important to notice that we do not explicitly consider modeling of time. This is because we do not know (nor care), *when* events in a system occur. Consequently, we lack of corresponding fine-grained concepts and relationships to repesent concrete time values. More important to ensure system consistency is to know, to which temporal context structural entities are associated and how these contexts are related to each other (see next section).

Representing User Interface

To complete our introduction of the most important concepts and relationships of our upper ontology for software models, we consider the representation of user interfaces. This relates nicely to the main categories of modeling languages in the classification scheme shown in Section 6.1.1.2. Now, as we stated there, parts of the user interface uniformely denote particulars, as they simply represent the UI that is shown during the execution of a software application. Moreover, without a doubt such elements do not denote temporal entities, as they do not specify any behavioral relationships. Thus, UI elements must be Actualitys. However, they are neither data structures nor any property instances.

In fact, user interface elements *represent* different parts of the system. Although they themselves denote Actualitys, they perfectly well might represent also temporal elements of a system. For example, consider the balloon tooltip, common to most operating systems, which occurs when particular system events arise, such as an unplugged network cable. The tooltip window, thus, would represent dedicated Events. To solve this dilemma, we introduced the concept of Representative, which simply represents different parts of the system, as depicted in Figure 6.14. This includes both, Universal and Particular entities, and Sets.

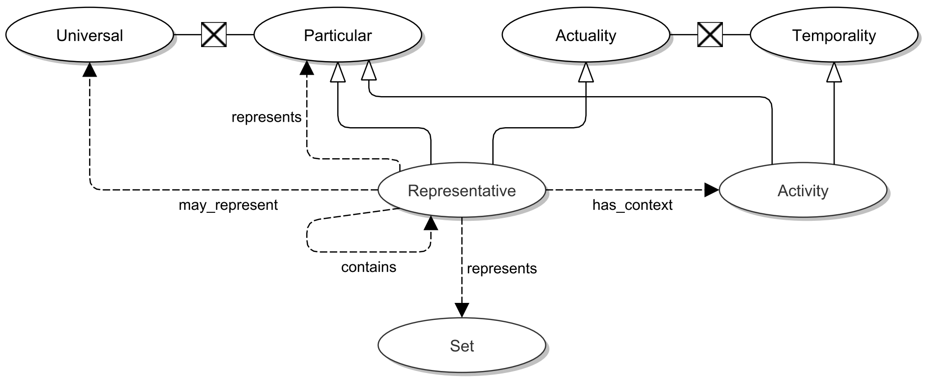


Figure 6.14: *The concept of entity representation in USMO*

Finally, one special type of relationship that we use in our upper ontology is important to highlight. The reader might have noticed the oddly named may_represent property between Representative and Universal. This kind of property is employed to establish *modal relationships* between Particulars and Universals. Including appropriate consistency rules, this enables us to simulate modal logic with DL-based Semantic Web languages. For example, if a Representative x may_represent a Univeral y, than it is allowed that x represents the Substantial z, if z instantiates (i.e., is_instance_of) y. More formally, expressed with the modal operator for *Possibly* $\Diamond$, this gives

$$\mathsf{may\_represent}(x, y) \leftrightarrow \Diamond\exists z\ (\mathsf{Substantial}(z) \wedge \mathsf{is\_instance\_of}(z, x) \wedge \mathsf{represents}(z, y)) \qquad (6.1)$$

The use of such modal relationships enables to establish semantic sound references between type and token models. This allows us, for instance, to associate UML class definitions with corresponding statecharts.

6.1.2.2 Selected Design Examples

After the introduction of fundamental concepts and relationships of USMO, this section will provide examples for the representation of common design constructs to be found in software models.

Stateful Behavior for Types

As introduced in the section above, stateful behavior is represented around the Life concept. To associate such behavior with corresponding type defintions, another modal relationship, the may_live property, comes into play (see Figure 6.15). This property relates corresponding type definitions with appropriate lifecycle behavior, probably specified with some statechart language in the modeling landscape of a software product. Besides this basic relation, the remaining depicted properties in Figure 6.15, reveal our approach to model the change of entities over time.

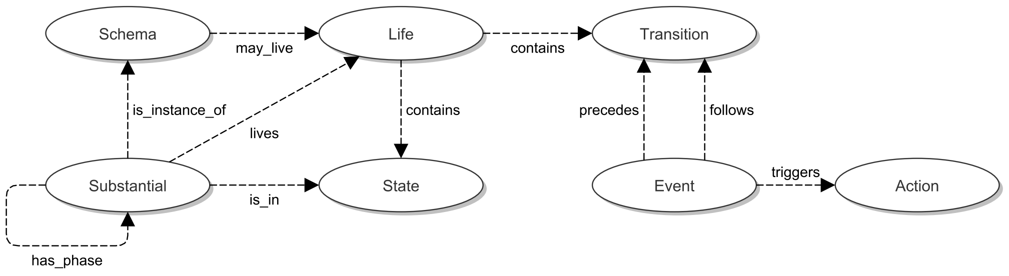


Figure 6.15: *Associating types with stateful behavior*

Here, we follow the approach proposed in ABC and GOL/GFO, which prefer the representation of an individual, i.e., a Substantial, with a universal and various existential facets. To this end, we define the property has_phase, which allows to associate these facets represented by different *ontically connected* Substantials. These existential facets are then connected with corresponding lifecycle States, via the is_in relation. In Figure 6.16 a concrete example is depicted that stems from the illustrative example, introduced in Section 4.3.

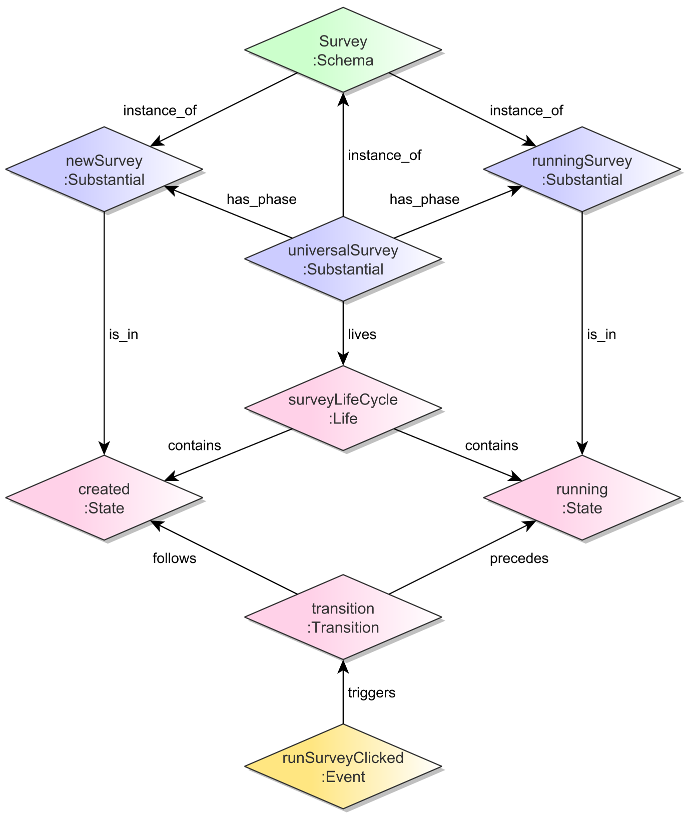


Figure 6.16: *Representing the change of entities over time*

The figure shows different indivuals as instances of particular USMO classes. Recall from Section 4.3, that we employed a Data DSL, a Service DSL, and a Form DSL to specify an application for the development of online surveys. Now, the green diamond in Figure 6.16 depicts the individual that is mapped to the corresponding Survey type, modeled with the Data DSL. The purple diamonds represent parts of a (in the example not considered) statechart language, to express the lifecycle of surveys. The blue diamonds do not directly map to existing modeling assets but are rather created to bridge between different DSMs. Thus, they describe that instances of the Survey type are comprised of a universal facet (universalSurvey), and two existential facets (newSurvey and runningSurvey). These, in turn, are related with appropriate states of the overall lifecycle for surveys.

Service Invocation

Another example, equally taken from the scenario of Section 4.3, can be given for the representation of service definitions and their invocation, respectively. The corresponding USMO types, to this end, are illustrated in Figure 6.17. As introduced above, we designated the concept Behavior to represent services. Behaviors are instantiated by Actions, which are likely to involve, create, or destroy particular Substantials. Services may be configured with dedicated parameters. To this end, USMO introduces appropriate temporal properties on type and token level (Temporal Property and Temporal Quality).

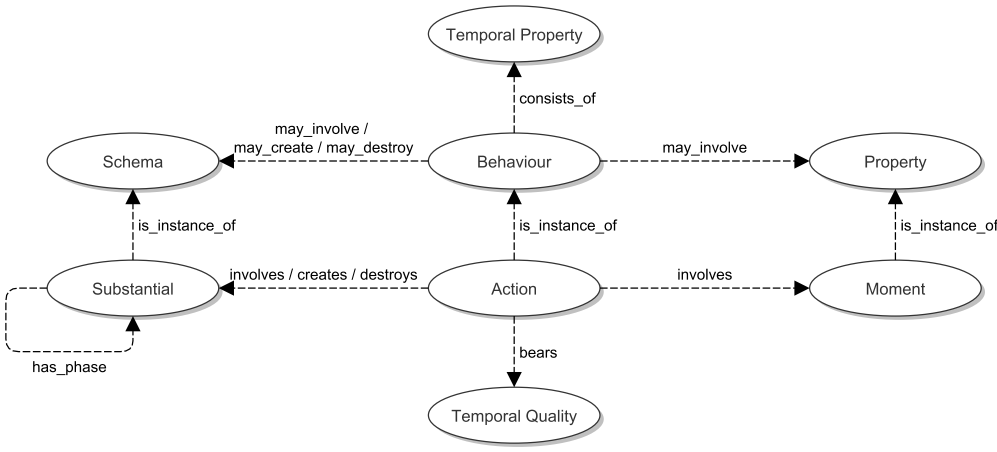


Figure 6.17: *Associating types and tokens with services and operations*

A potential excerpt from the knowledgebase of our illustrative example is shown in Figure 6.18. Here, we additionally depicted parts of the user interfaces, which trigger corresponding behavior. The figure shows the Representative saveButton, which maps to the corresponding form widget, and the Signal saveBtnClick, the kind of event the button actually represents. A click on the button yields an Event instance, which triggers the save survey Action. This, in turn, is instance of the Behavior, which maps to the corresponding service of the Service DSL.

6.2 Code Composition

The HybridMDSD approach facilitates a maximum level of reuseability through the ecapsulation of domain-specific modeling assets in language modules (cf. Section 4.2.2.4). Such modules include DSL metamodels, their semantic interpretation, model editors, interpreters, and generators that produce actual runtime code. Now, to facilitate end-to-end multi-domain engineering, it is not enough to integrate languages on modeling level, like it is possible with the central

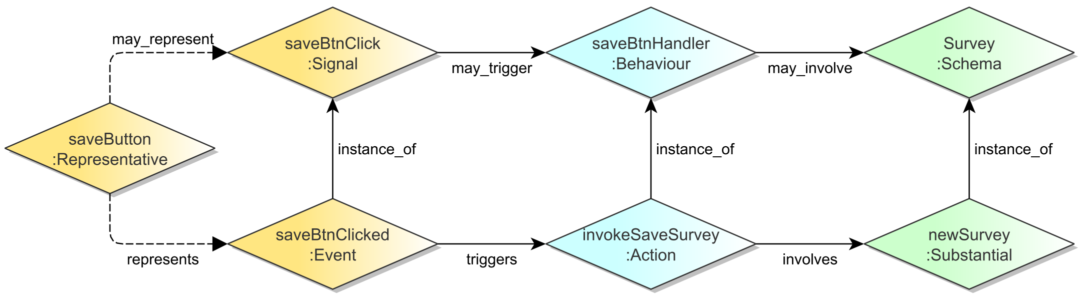


Figure 6.18: *Representing user interface event handlers*

upper ontology USMO (see previous section). It is rather necessary to integrate also the actual execution semantics of incorporated DSLs. This implies that the "executing facilities" of DSLs, i.e., language interpreters or code that is generated from corresponding DSMs, need to be composed as well.

In this section, we shed light on language composition at *code level*. Thus, we focus on the integration of DSL execution semantics on the basis of generated code assets. In Section 6.2.1, we explain why we limit our considerations to this kind of integration and provide necessary, preliminary context information. In Section 6.2.2, we give detailed explanations about the structure and functionality of the glue generation framework which implements the code composition task.

6.2.1 Preliminaries

Before we descend into the details of the HybridMDSD code composition component, this section spans the context for corresponding explanations. To this end, we clarify what the term *generator* eventually means in our context (see next section), discuss why HybridMDSD focuses on generative rather than interpretive MDSD scenarios in Section 6.2.1.2, and classify important code asset types in Section 6.2.1.3.

6.2.1.1 The Generator

The production of programming code from corresponding models is performed by *generators*. More specifically, a generator is

> "a program that takes a higher-level specification of a piece of software and produces its implementation" [CE00, p. 333],
> they are
> "development tools that automatically select, schedule, and apply black-box transformations to produce implementations from input specfications" [GS04, p. 505]

In the context of HybridMDSD, we are in line with these definitions and add that the "implementation" mentioned above, is not restricted to assets that contain code written in a particular programming language. Instead, produced assets may perfectly well comprise non-code assets, such as XML configuration files, internationalization resource bundles or human-readable documentation. From a technical point of view, we additionally perceive the set of templates that produce the desired output for a particular target platform including all related assets, like utility methods or runtime infrastructure, as a *self-contained unit*. This forms the actual generator in HybridMDSD.

In this context, the criterion of self-containedness also counts for the assets that are generated. Here, we are in line with Kelly and Tolvanen who state that "generated code is typically complete [...], executable, and of production quality" [KT08, p. 80]. They mention that the generated code should not need manual rewriting or additions, which is a central requirement for HybridMDSD generators too.

6.2.1.2 Interpretation vs. Generation

Code generation is only one possibility to use the information that is specified in domain-specific models. As introduced in Section 2.1.3.3, an alternative is the direct interpretation of this information in a particular execution environment. To this end, appropriate interpreters need to be available at runtime of the target software application. These interpreters read the input specifications and translate them directly into execution environment commands. The HybridMDSD approach limits the task of execution semantics composition to *generative scenarios*, i.e., such scenarios where programming code is produced prior to the runtime of a software application. We do so mainly for the following two reasons:

1. **Broad applicability:** Code generators are applicable to arbitrary target platforms without any limitations because the template developer has the freedom to produce arbitrary assets.
2. **Performance optimization:** Generated code can be steadily evolved to achieve a maximum of runtime performance. To this end, it can be highly optimized omitting any superfluous statements. Interpreters, instead, require additional execution cycles to read and translate input specifications at runtime.

Both, applicability and efficiency are necessary prerequisites for the use of HybridMDSD also in legacy systems. Software companies usually maintain a huge technology stack, including frameworks, libraries, or middleware platforms. Such technology stacks are commonly hard to adapt, exchange, or extend because of dependencies to published software and ongoing consumer support. This matters especially when platform-foreign technologies need to be introduced, e.g., to parse input models of a particular modeling framework. Nevertheless, the integration of language execution semantics with interpreters remains an interesting challenge and is subject of future research.

6.2.1.3 Glue Code

The integration of generated code produced from arbitrary DSLs necessitates the involvement of integrating code structures. Such code structures are called gluing or glue code [Aßm03, dNR04]. Even an appropriate *GoF (Gang of Four)* design pattern — the *mediator* — exists, which describes how to encapsulate collective behavior to implement the interaction of components [GHJV95, pp. 273-282].

However, in context of HybridMDSD we slightly extend this definition of *glue code*. To this end, we do not restrict its meaning to explicit structures that help to mediate between components but rather state that also small code fragments (which are woven into already generated programming code) need to be counted as glue code. This consideration is important with respect to the realizing glue generation framework, introduced in Section 6.2.2, because it properly distinguishes what types of integration fragements are to be produced.

In particular, we distinguish *implicit* and *explicit* glue code, and *composition* code, as illustrated in Figure 6.19. In this regard, the terms "implicit" and "explicit" refer to self-contained programming code that was generated from various DSMs of a particular DSL (cf. Section 6.2.1.1). Thus, *implicit glue code* denotes programming code that is embedded in generated

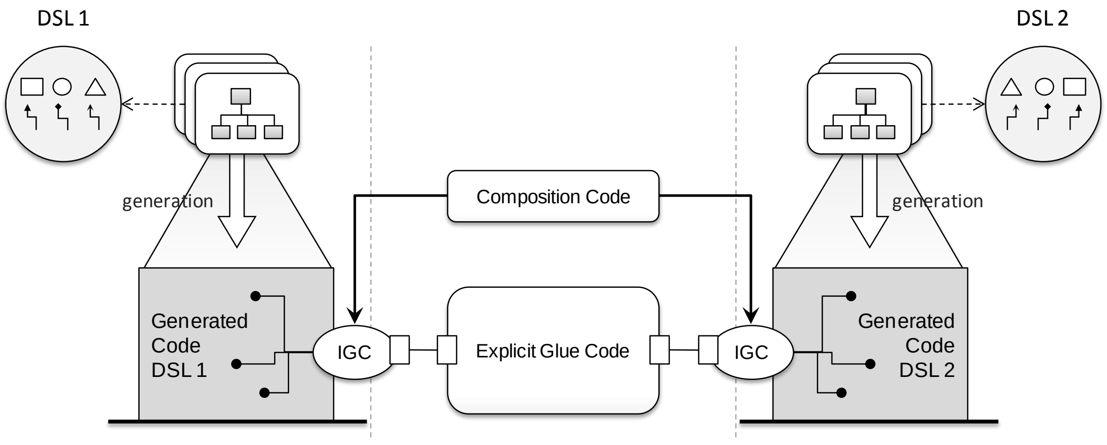


Figure 6.19: *Different glue code types in HybridMDSD*

code, e.g., through invasive techniques (see next section). *Explicit glue code*, in turn, is gluing that is particularly provided to mediate between generated code assets or that is at least part of such mediating structures. Hence, this kind of glue code matches with the original definitions stated above. Lastly, *composition code* identifies gluing code structures that are themselves composition recipes specified in a particular composition language and that realize the composition of dedicated code fragments (cf. next section).

6.2.1.4 Invasive Software Composition

In [Aßm03], Aßmann generalizes the problem of software composition and provides a comprehensive framework for involved entities and processes. In addition, he extends the common perception of component-oriented software development and explains how to go beyond the composition of black box components with properly designated interfaces. To this end, he presents the principle of *Invasive Software Composition (ISC)*, which defines *software components* and *composition interfaces* in terms of variably grained *fragment boxes* and *hooks*.

Fragment boxes are comprised of a set of program elements and a composition interface, which, in turn, consists of a set of hooks. Fragment boxes differ from common software components with regard to the applied composition type, because they are components that are subject to *invasive composition* [Aßm03, p.112].

Hooks identifiy positions in a program that are subject to change and, hence, designate variation points in programs.

Based on these definitions, Aßmann explains invasive software composition as a process wherein "[...] composers instantiate, adapt, extend, and connect fragment boxes by transforming their hooks" [Aßm03, p. 112].

Similarities with HybridMDSD

A consideration of the HybridMDSD code composition problem from the perspective of ISC, reveals a number commonalities. Like in invasive software composition, the employment of "conventional" software components with properly defined composition interfaces (cf. [Aßm03, pp. 64-101]), which can easily be integrated using simple black box composition, is rather the exception. Instead, the generative nature of HybridMDSD allows to produce arbitrary output that forms rather blurred components or, in other words, fragment boxes. In these *generated* *fragment boxes*, we too need to compose different assets based on dedicated hooks, which differ as explained in the following.

Code hooks are *code elements*, i.e., parts of the abstract syntax of the underlying programming language, that denote placeholders for other syntax elements. For instance, the type parameter in a generic class is a code hook.

Position hooks identifiy *positions* in a program that are subject to change. An example is the `extends <class>` statement of a Java class declaration, which is located between the actual class declaration and the curly bracket that opens the class definition.

Implicit hooks are either code or position hooks that are "contained in every component by definition of its programming language" [Aßm03, p. 116]. Prominent examples from the aspect community [KLM+97, Lad03, CEK03], are execution points in an application, where an advice may be applied, e.g., *before* or *after* the execution of dedicated methods.

Declared hooks are either code or position hooks that have deliberately been defined as subject to change. For instance, the above stated type parameter of a generic class or through structured comments that mark up dedicated code regions, are declared hooks.

In the chapter "Architecture as Composition" [Aßm03, p. 189], Aßmann explains the application of invasive software composition to facilitate the binding of communication partners and illustrates how *composers* act as *connectors*. In this regard, he describes use cases where these connectors produce an explicit gluing infrastructure that is responsible for communication [Aßm03, p. 200, Table 7.2, Figure 7.3]. This is very related to what we do with HybridMDSD, because architectural composition is most likely a prominent use case in HybridMDSD projects. That is because domain-specific languages are created for dedicated application domains and important domains, such as user interface, data or process modeling, usually result in particular architecture components. These, in turn, need to be integrated with appropriate connectors, like described in ISC. Additionally, the production of gluing infrastructure code is in line with the different glue code types, we identified in the previous section, and matches with our composition approach, as explained later in Section 6.2.2.

Differences to HybridMDSD

However, there are also few important differences between HybridMDSD code composition and bare ISC. Although we definitely consider the code composition task in this work as an application of invasive software composition, we underlie some concrete requirements that did not need dedicated attention in the general description of ISC by Aßmann.

First, the generative scenario of HybridMDSD imposes the challenge to compose *arbitrary platform implementations*. Thus, information about hooks in generated code needs to be mapped to information about the particular platform, i.e., the programming language or specification schema the code adheres to, in order to facilitate proper interpretation of these hooks. A position hook, for instance, pointing at a line of code in a Java class, can be interpreted as code hook denoting a particular parameterizable method declaration, only if the necessary information about the employed platform, such as Java Standard Edition [Mic09b], is available. This is an addition to ISC, where Aßmann admittedly explains that "fragment boxes can be identified in any programming or specification language whatsoever [...]" [Aßm03, p. 115], but does not further consider the impact of multiple platforms on the complexity of composition operations.

Second, the model-driven approach of HybridMDSD imposes the significant challenge to derive a proper mapping between metamodels, models, and generated code. So far, the ontological foundation of the HybridMDSD architecture allows for the creation of interconnections on modeling level only (cf. Section 4.2 and 6.1). The integration task on code level, thus, needs to be aligned to (1) *basically possible connections* on M2 level (provided by mapping rules to language ontologies) and (2) *actually instantiated connections* on M1 level (provided by the integrating project knowledgebase). Hence, for a particular model element, and its metamodel type respectively, we need to derive dedicated hook information, in order to facilitate an integration on code level. Although this is in a sense a preliminary task to the actual composition operation, it is a major prerequisite for the composition of arbitrary platforms and delimits our work from ISC.

Last, also because of the challenge of platform integration, the composition technique always depends on the kind of the platforms that need to be integrated. In Java, for example, the invasive integration of additional code would be possible either with aspect languages, such as AspectJ [Lad03], or dedicated composition tools, such as Aßmann's prototype for invasive composition *Compost* [Aßm03, p. 121]. However, for the integration of programming code written in C++ or even declarative statements serialized in XML syntax, other composition tools are necessary. It might even be possible to fall back on untyped string replacement operations to manipulate source code artifacts, in case employed programming languages do not provide necessary means for manipulations, such as introspection. Hence, not only the interpretation of hook information depends on the employed plaform, but also the kind of applicable composition techniques and languages.

Conclusion

As the code composition component of HybridMDSD denotes an actual composition system, we want to class our approach with the definition of composition systems given in [Aßm03]. Thereafter, composition systems are comprised of the following entities:

A Component model which specifies how a component and its composition interface in the system should look like.

A Composition technique that defines how components should be composed, i.e., what means are available for composition. Examples are glueing or aspect separation (cf. [Aßm03, p. 19]).

A Composition language that is used to specify *composition recipes*, which automate the composition process in a system. *Architecture Description Languages (ADL)* are prominent examples for composition languages.

Searching for these ingredients in the HybridMDSD code composition approach, leads to the following results. The *component model* of HybridMDSD coincides with that of invasive software composition. As stated above, the flexible concept of fragment boxes and hooks should be applicable to arbitrary languages, which renders it to an ideal candidate for the generative scenario we are dealing with in our context. In contrast, *component technique* and *language* are rather difficult to identify. This is because, each programming language that is subject to integration, necessitates an appropriate composition system that provides a corresponding technique and language. Thus, we do not aim at designing a one-size-fits-all composition system, which facilitates the composition of arbitrary languages using a comprehensive, language-independent composition technique. We rather apply a set of available composition systems that are appropriate for particular use cases. In this respect, the loose component information from HybridMDSD needs to be translated into use case-specific component models of the applied composition system (see Section 6.2.2.2).

6.2.2 Glue Generation Framework

In the previous section we clarified important terminology, justified the limitation to code composition scenarios, and discussed the alignment of our approach to the principle of invasive software composition. This section presents the *Glue Generation Framework (GGF)*, which finally realizes the integration of generated source code from different DSMs. We provide an overview in Section 6.2.2.1, enlarge upon the challenge of composing arbitrary platforms in Section 6.2.2.2, and shed light on the actual composition process in Section 6.2.2.3.

6.2.2.1 Overview

Figure 6.20 gives an overview of the HybridMDSD code composition framework. The illustration is a refined version of the general architecture overview given in Section 4.2.2 (cf. Figure 4.6) that focuses on code composition.

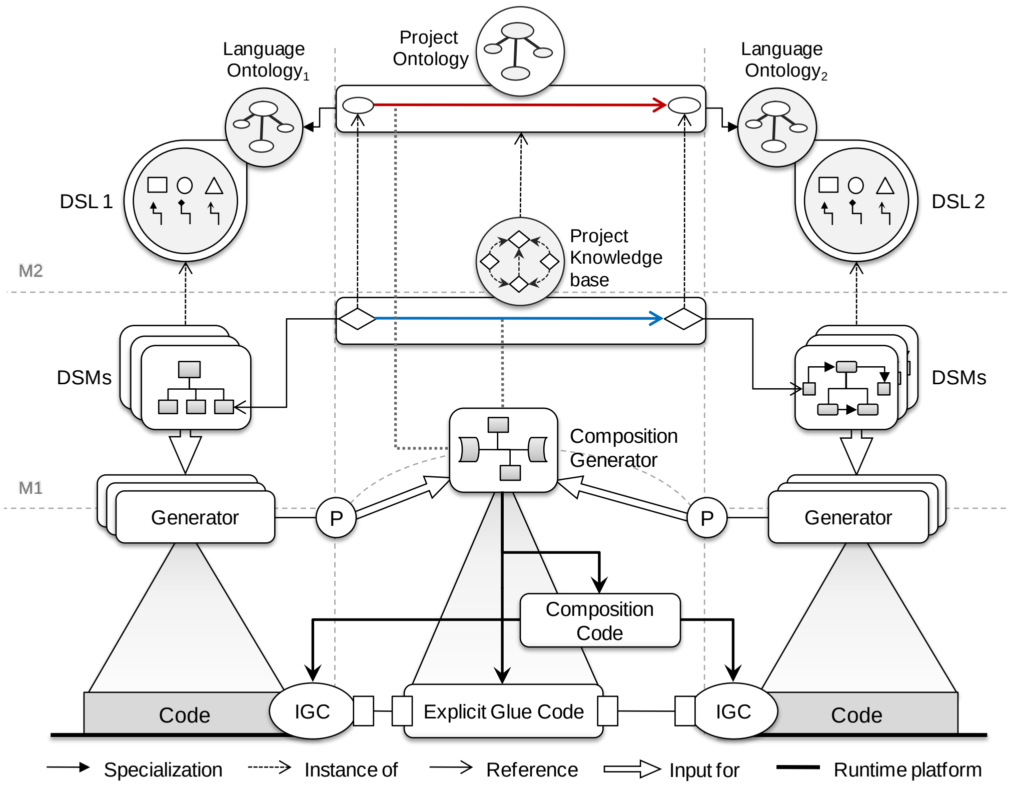


Figure 6.20: *Overview of the HybridMDSD code composition architecture*

As in the general architecture overview, Figure 6.20 illustrates language composition in HybridMDSD using the example of two DSLs, depicted on the figure's left side and right side, respectively. The layered illustration shows from top to bottom three levels of abstraction, meta level M2, meta level M1, and code level. The figure's centre shows the integrating project ontology on level M2 as well as the instantiating project knowledgebase on level M1. In particular, a concrete inter language connection is illustrated on both levels.

The centre's lower part depicts the *composition generator*, which is the central component in the GGF. Attached to a particular inter-language connection, the composition generator contains the composition recipe for a dedicated composition scenario and produces implicit, explicit, and/or composition code tailored to this scenario. Composition generators are reusable and must be specified *only once* for each inter-lanuage connection that needs to be integrated on code level. To this end, they depend on information about employed platforms and corresponding hooks, which is provided by dedicated *generator profiles* that are attached to platform generators (depicted as circular framed P; see next section).

6.2.2.2 Generator Profiles

As explained in the sections above, one major challenge when integrating arbitrary domain-specific languages on code level, is the potential employment of different technology platforms. This is challenging on different levels of abstraction. Thus, it is well possible to employ entirely different programming or specification languages in one project. A prominent example is the employment of declarative languages, such as XAML [Nay07] or JSF [Mic09d], to specify the user interface of applications, while using an imperativ language, such as Java or C#, to define its business logic. Another common use case is the employment of different frameworks or programming code libraries, even if the very core platform remains the same.

While fundamental differences in the underlying programming or specification language have an impact on the composition system that is applicable for the integration of glue code, differences in involved frameworks and libraries merely affect concrete composition code. In either case, appropriate tooling needs to provide information about the underying platform in order to facilitate and optimize composition. To this end, we use *generator profiles* that gather necessary data (see Figure 6.21).

Generator profiles comprise a static and a dynamic part. The static part encapsulates appropriate properties, identifying the underlying runtime platform. This information is necessary to select an appropriate composition system for a particular composition use case. The dynamic part is used to resolve corresponding hooks in the generated code, given a particular modeling construct. This hook resolution tooling can be used when specifying the composition generator for a dedicated inter-language connection.

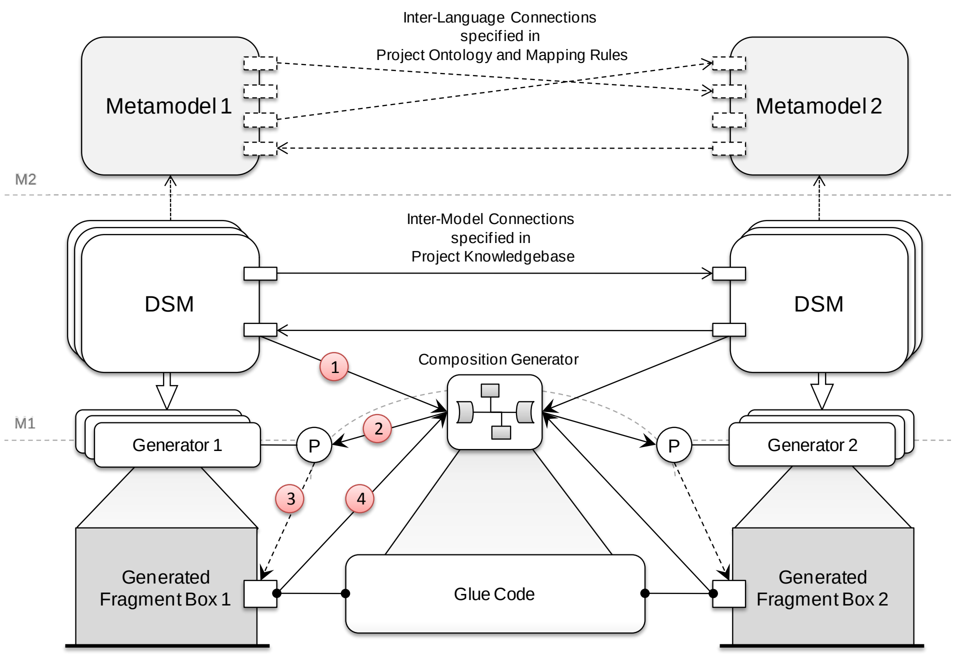


Figure 6.21: *The role of generator profiles in the HybridMDSD code composition*

Hook Resolution

Having clarified the role of generator profiles in context of the resolution of model element manifestations in programming code, we briefly elaborate on this resolution process, which is illustrated in Figure 6.21. The figure shows how the actual language composition is realized across the different levels of abstraction. On level M2, connections between languages are established through project ontology definitions and the mapping rules that implement the instantiation of knowledgebase individuals during modeling. On level M1, corresponding knowledgebase individuals persist and reference related model elements.

Given a particular connection between two domain-specific models, the generator profile needs to resolve related code manifestations. To this end, during the composition process (see next section) the composition generator (1) receives source and target elements and (2) requests code manifestations from the generator profile. This, in turn, (3) calls manually implemented resolution methods and (4) returns corresponding hooks to the composition generator.

6.2.2.3 Composition Process

In this section, we complete the presentation of the HybridMDSD code composition framework with an explanation of the actual composition process, which is executed succeeding to the generation of domain-specific programming code from corresponding DSMs. The actual composition algorithm is shown in Listing 6.1.

```
for (LanguageConnection languageConnection : projectOntology
        .queryLanguageConnections()) {

        CompositionGenerator compositionGenerator;
        compositionGenerator =
                queryCompositionGenerator(languageConnection);

        for (ModelConnection connection : projectKnowledgeBase
                .queryModelConnections(languageConnection)) {

                Composer[] composers = compositionGenerator
                        .generateComposers(connection);
                for (Composer composer : composers) {
                        composer.compose();
                }
        }
}
```

Listing 6.1: *Composition execution algorithm in HybridMDSD*

As listed, the composition process in principle follows the layered architecture, often depicted in previous figures (cf. Figure 6.21, 6.20, or 4.6). The individual tasks are the following:

1. All possible inter-DSL connections are queried from the central project ontology.
2. For each connection, the dedicated composition generator is requested.
3. The knowledgebase is queried for all present inter-model connections.
4. The composition generator is invoked for each connection, to produce implicit glue code, explicit glue code, and/or composition code, which employs above introduced extended hooks. The produced code forms the actual *composers*.
5. In case generated glue code denotes composition code, corresponding composers need to be adviced to perform the composition.

The algorithms that are called in Listing , are explained in the following:

`queryLanguageConnections()`

This algorithm queries and returns all inter-language connections that exist in the underlying development project. A language connection is usually represented by a particular ontology property that interconnects concepts of different language ontologies. But, also more complex connections are possible, as shown by example in Section 7.2.3.5.

`queryCompositionGenerator(LanguageConnection lc)`

This algorithm queries and returns the composition generator that is attached to the passed language connection. An example for such a composition generator is given below.

`queryModelConnections(LanguageConnection lc)`

This algorithm queries and returns actual model connections, i.e., inter-model references, that instantiate the given language connection.

`generateComposers(ModelConnection mc)`

This algorithm invokes the composition generator which then produces all composers for the passed inter-model reference.

`compose()`

This algorithm invokes the called composer which finally executes composition.

Example Scenario

To provide a better understanding of how composition generators and glue code actually are characterized, the Listings 6.2 and 6.3 illustrate exemplary assets. Assumed is a scenario where two language modules produce Java programming code from model instances. This Java code needs to be composed. For composition, an advanced system is necessary, which facilitates the introduction of implicit glue code into generated assets. Here, AspectJ [Lad03] is used as composition system, to perform this task.

Listing 6.2 shows an appropriate composition generator for the scenario. It shows a generator template written in the Xpand template language provided by the generator framework openArchitectureWare (oAW) [ot09]. As listed, the composition generator defines a non-typed template (indicated by `Void` at the end of the `DEFINE` statement), which is parameterized with source and target types appropriate for the corresponding language connection `has_statemachine`. Thus, in the present case, a connection between a data modeling language and a behavior modeling language is to be implemented.

To this end, a `BusinessObject` is connected with a `StateMachine`. The generator template advices the generator engine to produce an AspectJ aspect file, named by corresponding identifiers for the language connection, the source model element, and the target model element. Here, it reveals how extended hooks are queried from the generator profile. Thus, the method `id(source)`, for instance, returns an identifier for the passed `BusinessObject` model element. This, in turn, identifies the class name the relates to the actual element, because the generator produces one Java class for each of these elements.

The finally produced composition code for a particular model connection is shown in Listing 6.3. The presented connection was drawn between a `BusinessObject Quotation` and the `StateMachine QuotationLifeCycle`. The snippet shows how extended hooks of corresponding model elements resulted in appropriate class names, which were returned by the generator profile. For brevity reasons, we omitted full qualified class names and assume proper import statements in the aspect's. As illustrated, the present aspect introduces proper attributes in each of the composed classes and adjusts the value of the `state` attribute of the `Quotation` class after an event was triggered in the state machine.

```
«IMPORT de_feasiple_salesscenario_businessobject»
«IMPORT de_feasiple_salesscenario_state»
«EXTENSION de::feasiple::salesscenario::hmdsd::has_statemachine::id_prvdr»
«EXTENSION de::feasiple::salesscenario::util::ext::util»

«DEFINE aspect(BusinessObject source, StateMachine target) FOR Void-»
«FILE getFileForPackage("de.feasiple.salesscenario.has_statemachine",
        "has_statemachine_" + id(source) + "_" + id(target), "aj")-»
package de.feasiple.salesscenario.has_statemachine;

import «source.bundle.name-».«source.name-»;
import «target.namespace-».«target.id-»;

public aspect has_statemachine_«id(source)-»_«id(target)-» {

        // statemachine into businessobject
        private «id(target)-» «id(source)-».lifecycle =
                new «id(target)-»(this);

        public «id(target)-» «id(source)-».getLifeCycle() {
                return source.lifecycle;
        }

        // change invocation
        pointcut getState(«id(source)-» «argument(source)-»):
                target(«argument(source)-») &&
                execution(public * «source.name-».getState());

        String around(«id(source)-» «argument(source)-»):
                getState(«argument(source)-») {
                return «argument(source)-».lifecycle.getCurrentState();
        }

        // businessobject into statemachine
        private «id(source)-» «id(target)-».«argument(source)»;

        public  «id(source)-» «id(target)-».getQuotation() {
                return this.«argument(source)-»;
        }

        public «id(target)-».new(«id(source)-» «argument(source)-») {
                this();
                this.«argument(source)-» = «argument(source)-»;
        }

        // change invocation
        pointcut fireEvent(«id(target)-» «argument(target)-»):
                target(«argument(target)-») &&
                execution(public * «id(target)-».fireEvent(*));

        after(«id(target)-» «argument(target)-»):
                fireEvent(«argument(target)-») {
                «argument(target)-».«argument(source)-»
                        .setState(«argument(target)-».getCurrentState());
        }
}
«ENDFILE»
«ENDDEFINE»
```

Listing 6.2: *Examplary meta-composer producing aspect-oriented compositions for dedicated language connection*

```
package de.feasiple.salesscenario.has_statemachine;

// ...
public aspect has_statemachine_Quotation_QuotationLifeCycle {

        // statemachine into businessobject
        private QuotationLifeCycle Quotation.lifecycle =
                new QuotationLifeCycle(this);

        public QuotationLifeCycle Quotation.getLifeCycle() {
                return source.lifecycle;
        }

        // change invocation
        pointcut getState(Quotation Quotation): target(Quotation)
                && execution(public * Quotation.getState());

        String around(Quotation Quotation): getState(Quotation) {
                return Quotation.lifecycle.getCurrentState();
        }

        // businessobject into statemachine
        private Quotation QuotationLifeCycle.Quotation;

        public  Quotation QuotationLifeCycle.getQuotation() {
                return this.Quotation;
        }

        public QuotationLifeCycle.new(Quotation Quotation) {
                this();
                this.Quotation = Quotation;
        }

        // change invocation
        pointcut fireEvent(QuotationLifeCycle quotationLifeCycle):
                target(quotationLifeCycle) &&
                execution(public * QuotationLifeCycle.fireEvent(*));

        after(QuotationLifeCycle quotationLifeCycle):
                fireEvent(quotationLifeCycle) {
                quotationLifeCycle.Quotation
                        .setState(quotationLifeCycle.getCurrentState());
        }
}
```

Listing 6.3: *Examplary aspectual composer for dedicated model connection (as instance of corresponding language connection)*

6.3 Development Process

In Section 3.4 (see Requirement 1), we identified the need for development guidance during language creation. Besides this general MDE requirement, also the complexity of the proposed solution approach clearly demands for a dedicated description of the different phases and activities that need to be mastered, in order to leverage the significant software development improvements of HybridMDSD. To this end, we designed a comprehensive development process, which is presented in the remainder of this section.

The section starts with preliminary considerations (see Section 6.3.1) and a general overview of the designed process (see Section 6.3.2). Subsequently, we present detailed explanations for particular development phases in the sections 6.3.3, 6.3.4 and 6.3.5.

6.3.1 Preliminaries

In this section, we elucidate the main principles we followed when designing a development process for HybridMDSD. The section starts with a brief historical wrap-up of software development processes, exposing the core values of modern agile development methods, which we mostly share (see Section 6.3.1.1). In combination with subsequently introduced approaches to model-driven development processes (see Section 6.3.1.2), we use these values to derive design principles for our own process definition (see Section 6.3.1.4). In this respect, we explicitly highlight the importance of use-case based evaluation throughout the entire process.

6.3.1.1 On the Way to Agile Development

When software engineering was in its infancy, development activities happened in a rather unstructured and chaotic manner and the success of software projects was solely decided by the skills and experiences of involved developers. The prominent *waterfall model* was the earliest approach to establish control structures and to counteract the ad-hoc development of software [WV07]. Introduced by Royce in 1970 [Roy70], it prescribes a linear process model that strongly emphasizes the importance of upfront, detailed planning in order to achieve predictability in development. Another contrasting early approach to software development is the *iterative model*, firstly introduced in 1975 [BT75]. It advocates the idea of developing software incrementally, continuously releasing deliverable system versions and allowing the developer to incorporate leassons learned from previous versions. With the *spiral model*, Boehm presented an additional iterative approach in 1986 [Boe86].

The iterative model has recently gained a lot of attention, with the *Agile Software Development (ASD)* movement. As a response to the rapdly changing requirements and the fast moving technology evolution, several "light-weight" methods emerged, which should help to overcome the inefficiency of conventional software development methodologies [WV07]. Among these, the probably most prominent are Extreme Programming (XP) [Bec99] and Scrum [Mur04]. In principle, agile development questions the values of detailed and comprehensive project planning and advocates the production of software "in a lighter, quicker, more people-centered way" [SR08]. Agile development originated at a meeting of a group of scholars and practitioners in 2001, who agreed on core values and guiding principles in the "Manifesto of Agile Software Development" [BBB+].

The agile manifesto constitutes four major values that are contrasted with other, competing values. The core values are

1. *Individuals and interactions* over processes and tools
2. *Working software* over comprehensive documentation
3. *Customer collaboration* over contract negotiation
4. *Responding to change* over following a plan

Hence, agile software development preaches to avoid rigorous and rigid plan-driven development and fosters communication between stakeholders, early prototyping, and iterative development. Within HybridMDSD, we follow these values in principle, like explained in Section 6.3.1.4.

6.3.1.2 Process Approaches to Model-Driven Development

In literature, only few examples for comprehensive process descriptions that are entirely dedicated to model-driven software development, including language design, generator/interpreter development, and language application, can be found [Bet04, SVEH07]. Instead, much work has been done towards the creation of proper domain-specific languages [EJ01, Coo06, Vis08]. For example, Spinellis collected a number of recurring design patterns for DSL development [Spi00]. Guizzardi et al. approached the challenge of language design from the perspective of theoretical foundations [GPvS05]. Other publications provide experience reports for DSL development with particular tool suites [LKT04, JBC$^+$06]. Moreover, several authors mix process aspects with other technologies, such as SPLE or the development of domain-specific frameworks [CE00, GNR04, Bet04, Cza05]. For example, Bettin and Grant divide model-driven software development into domain and application engineering which is common for product-line engineering (cf. Section 2.3.3) [GNR04, Bet04].

To this end, Grant et al. count the creation of models to the application engineering phase. In this thesis, we do not follow this consideration and argue that application engineering comprises activities that are specific to *particular* software products (cf. Section 2.3.3). Instead, in a software product line the complete set of specification models is reused and modified for *many* software products. The modeling landscape is created to provide actual domain assets that are reused in application engineering. Thus, modeling is to be counted to the domain engineering phase of development.

In summary we can say that existing process approaches to MDSD do not entirely cover all aspects involved in multi-domain engineering with HybridMDSD. Nevertheless, existing work is very valueable to elaborate the distinct phases in development, as presented below.

6.3.1.3 Use Case-Based Evaluation

As very useful means for effective development of various assets around model-driven technologies, including DSLs, editors, generator, or model transformations, proved small use case-based evaluation scenarios that document on the general application of the assets to develop. The employment of such scenarios as active development means is common to MDSD, because they facilitate efficient and straightforward assessment of the scope and appropriateness of domain-specific concepts in both, abstract metamodels and implementing generators or interpreters [SVEH07].

To this end, corresponding scenarios can be delivered as *reference model*, *reference implementation*, or *example application*, respectively [SVEH07, p. 222][Bet04]. Reference models are mainly used to assess the accuracy of domain-specific languages. They test whether typical usage scenarios of the domain of discourse can be represented with available language constructs. Reference implementations or example applications, instead, are used to capture the semantics of an abstract language. They denote representative implementations of the domain of discourse, again for a dedicated usage scenario.

In the conext of our work, we advocate the use of both, simple straightforward reference models and reference implementations likewise. Reference models are crucial for early evaluation of conceived DSL design in every development iteration. Reference implementations are well suited to derive domain requirements and, thus, assist during domain scoping. Moreover, they act as optimal starting points to develop generator templates, as they specify the finally necessary code generation outcome.

Concerning actual use case descriptions, we consider the specification of comprehensive use cases, including stakeholder descriptions, minimum guarantees, postconditions, business rules, and performance constraints, as inappropriate and superfluous. We argue that, above all because of the domain specificity, the general character of complex use case descriptions would span different solution domains of a single system. Instead, we follow Cockburn's notion of the "use case brief", which is a minimal sentence-based description of a usage scenario [Coc02] that is appropriately tailored to the domain of discourse. However, we do not convict the employment of more complex use cases, but emphasize that these should rather be used as basis for the extraction of several domain-specific use cases in language module development.

6.3.1.4 Design Principles

Multi-domain engineering facilitates development with multiple DSLs, improving a proper separation of concerns (cf. Section 3.3). This entails, that each concern is represented by one or more stakeholders that claim the implementation of individual domain-specific requirements. Basically, this is nothing new in software development. Also in "conventional" development, experts for dedicated solution domains, such as user interface designers or database specialists, need to synchronize their development efforts.

However, using abstract DSLs to specify running software, promotes the idea of separation of concerns to a higher level. The employment of model-driven technologies allows also non-technical but rather domain-specific experts to contribute directly to software applications. This strengthens the role of these domain-experts in the overall development process, as they become active participants of this process.

Althogh improved separation of concerns and abstract domain-specific development opens the door for a number of improvements (see Section 4.4), it also implicates several risks. The development of domain-specific assets, such as DSLs, appropriate editors, code generators, or model transformations, is costly. Although currently available tool suites and IDEs (see Section 2.1.2.5) evolve steadily and already provide impressive features for model-driven development, the management of *change* and *evolution* is still an unsolved problem.

Thus, the advantages of MDSD could easily turn into expensive drawbacks. In order to obviate the risk to this effect, we constitute general principles that shall guide us through the design of a HybridMDSD development process. These principles are mainly derived from the core values of agile methods and implications of MDSD.

Sustainable Development

The creation of domain-specific tools must be performed accurately to ensure that indeed *domain-specific* and *not application-specific* assets are created. HybridMDSD strongly advocates reuse of once created language modules. If created modules are not reusable, this intention can not be implemented. In order to achieve reusability, language modules need to be developed with respect to appropriate scoping, granularity, and practicability.

Awareness of Existing Assets

The organization of domain-specific assets in language repositories facilitates the problem-centric reuse of these assets. Of course, a development process for HybridMDSD needs to stress the use of these existing assets. If not, all improvements towards reusabilty and bundling of once created implementations is in vain.

Iterative and Incremental Development

As explained above, change and evolution are the most difficult challenges model-driven development faces today. To pave the way for sustainable development, an iterative and incremental approach is crucial. Early prototypes to evaluate DSL designs, editing tools, or code generators

generators are essential preliminaries to the actual modeling phase [SVEH07, p. 218][Bet04]. Late changes in metamodels, for example, at worst invalidate the modeling landscape and generator templates likewise and must definitely be avoided.

Communication and Feedback

We cleary emphasizes the need for communication between different stakeholders. On the one hand, stakeholder communication in context of MDSD should be easier compared to conventional approaches. Abstract models can be used as direct subjects for discussions, results can be directly incorporated, and the effect of changes is immediately visible either in models or in ad-hoc generated, executable software. On the other hand, pre-eminently during the creation of DSLs and corresponding platform implementation, communication is vital to yield accurately tailored domain-specific assets.

6.3.2 Process Overview

Before we explain the details of the HybridMDSD development process, this section provides an overview. We shed light on the activities to cover (see Section 6.3.2.1), the overall development phases (see Section 6.3.2.2), and participating roles (cf. Section 6.3.2.3).

6.3.2.1 Involved Activities

As a first step towards the design of a HybridMDSD development process we summarize general activities, which such a process needs to cover. In the following, we listed a number of activities that need to be covered by such a process. We derived these from (1) the general requirements for MDE and (2) the HybridMDSD architecure.

Language Design comprises the specification of metamodels, i.e., a language's abstract syntax, and appropriate visual or textual respresentations for instances of these metamodels, i.e., a language's concrete syntax.

Language Ontology Design comprises the selection of DSL constructs that are *relevant* for integration, the determination of the semantics for these constructs, and the development of appropriate mapping rules that instantiate this conceptual projection for each model.

Language Choice comprises query and selection of DSLs during the development of concrete projects, where multiple languages shall be used in combination.

Language Composition comprises the specification of inter-language relationships, constraints, and implementational composition structures, subsequently to language choice.

6.3.2.2 Development Phases

The above introduced activites can be grouped into three general activity types , namely *tool creation*, *tool composition*, and *tool application*. Recall from Section 3.3.2, that the notion of multi-domain engineering is derived from the product-line development phase "domain engineering", which comprises "development of a common architecture for all the members" [Cza05]. Hence, HybridMDSD focuses at establishing a domain infrastructure to (re-)produce software products and product lines. This is why we coin the base actitivites *tool-centric activities*, aiming at the creation of a tool infrastructure for development. In this respect, the term "tool" covers all types of assets that are employed for actual software development, in the given context. This comprises DSLs, code generators, ontologies, mappings, and integration code structures.

Table 6.2 depicts the different activity types and relates them to corresonding phases of our development process. Additionally, the outcome of each development phase is summarized.

Activity Type	Development Phase	Development Output
Tool Creation	Language Engineering	Language Module (incl. metamodel, model editor(s), generator(s), language ontology, mapping rules)
Tool Composition	Project Engineering	Modeling Workbench (incl. set of language modules, project ontology, code composition patterns)
Tool Application	Modeling	Modeling Landscape (incl. DSMs for integrated DSLs, generated code, generated composition code)

Table 6.2: *Different HybridMDSD development phases*

Thus, tool creation is performed during the *language engineering* phase, tool composition during *project engineering*, and during the *modeling* phase the established tool infastructure is finally applied to develop software. Figure 6.22 depicts how the different phases relate to each other. Recall from Section 4.2.2.4, that we organize reusable domain-specific assets in language modules. These modules are produced in the language engineering phase, aggregated to modeling workbenches in semantic connector projects during the project engineering phase, and finally used in the modeling phase to develop product (line) implementations.

As indicated in Figure 6.22, the HybridMDSD process might either start directly with language engineering or with the later project engineering. This is because of the option to start project development directly with language modules that are available from the language repository. As the arrow from project to language engineering shows, necessary language modules need to be created if these modules are not appropriate.

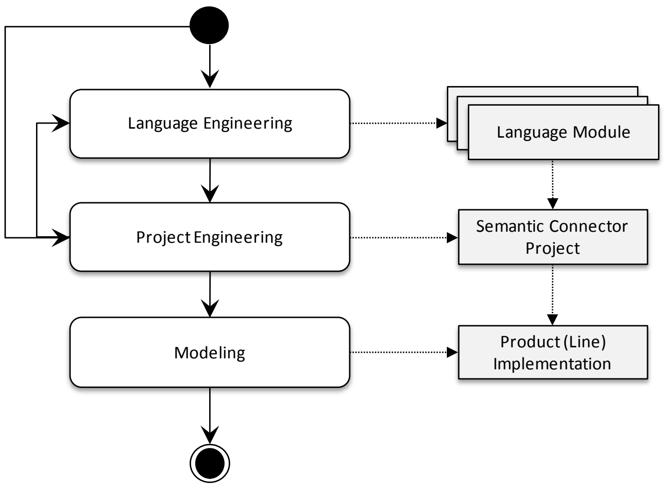


Figure 6.22: *HybridMDSD development process overview*

6.3.2.3 Development Roles

Taking a look at the HybridMDSD architecture quickly reveals that the approach involves a number of different technologies. Besides its general foundation, which is based on the paradigms of MDSD, DSM, and SPLE (cf. Section 3.3.2), it comprises Semantic Web and code composition technologies. Consequently, the number of development roles in the HybridMDSD process is not neglectable and currently counts seven different parts. However, as other approaches to MDSD or DSM prove, we do not involve unduly many development roles. In [KRV06], for example, Krahn et al. propose to subdivide the activities in domain-specific modeling between *four* parties, namely *language developers*, *tool developers*, *library developers*, and *product developers*. Also, Stahl et al. count *seven* diverse development roles likewise, excluding the customer role, as depicted in Figure 6.23.

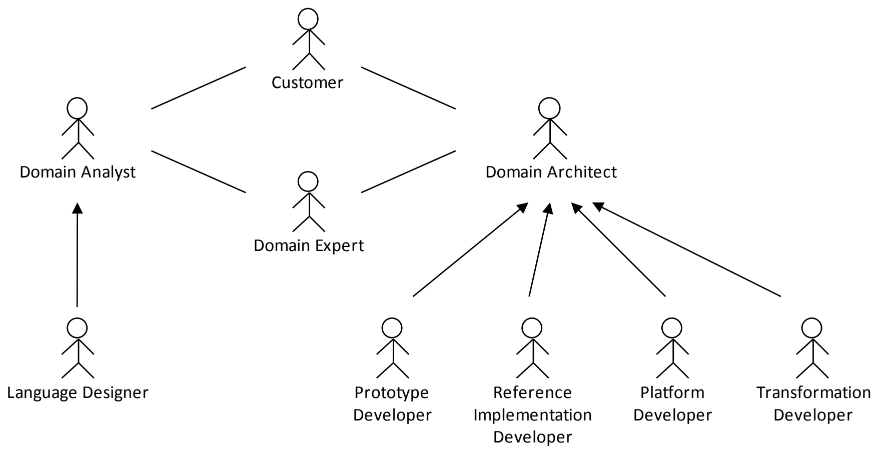


Figure 6.23: *Development roles in MDSD (adapted from [SVEH07, p. 344])*

Either way, the number of roles should not confuse. Today, commonly quite a number of the later enumerated HybridMDSD development roles are unified in few persons or even one single developer. This is because the nature of modeling languages and their respective meta-metamodels are naturally aligned to object-oriented paradigms [MOF06a, p. 11], which nearly all of todays developers are familiar with. Nevertheless, it is useful to clearly identify, name, and separate different engineering tasks in corresponding roles for MDE, in order to efficiently allocate workstreams and assign activities. In the following, we enumerate the development roles of HybridMDSD.

Domain Expert is a person with specialist knowledge about a particular domain of discourse. This includes both, technicians that rule dedicated solution domains in software development, such as user interface design or persistence specification, and non-technicians that are experts in the general business domain of a software product or product line.

Language Designer is a specialist in model-driven development, who is experienced in creating metamodels and appropriate editing tools. Together with the domain expert, the language designer is responsible for DSL development.

Knowledge Engineer is an expert for semantic questions on both, conceptual and technical level. Thus, this role unifies theoretical knowledge around terms like conceptualization, ontological commitment, or description logics, and practical experience in ontology design and with Semantic Web technologies (see Section 2.4).

Modeler is someone who specifies domain-specific models for a particular solution domain. This comprises, thus, usually experts for solution domains, who are experienced in software modeling.

Composition Expert is responsible for the technical integration, i.e., the development of composition patterns to form a comrehensive modeling workbench, also on code level. Composition experts, thus, come up with dedicated knowledge and practical experience with composition techologies, such as AspectJ [Lad03].

Platform Developer is a solution space expert, which is experienced with necessary know-how to create reference implementations and provide domain-specific frameworks form which generator templates can be derived.

Transformation Developer is experienced in using template and transformation languages of appropriate frameworks and APIs. Thus, the transformation developer is responsible to create the bridge between modeling and code level.

6.3.3 Language Engineering

The language engineering phase in HybridMDSD comprises activities around the development of domain-specific assets. This includes the creation of actual domain-specific languages (considered in Section 6.3.3.1), the specification of code generators (see Section 6.3.3.2), and, finally, the mapping DSL constructs to an appropriate language ontology (see Section 6.3.3.3). Figure 6.24 illustrates how the different activities interact, what assets they result in, and which development roles are involved.

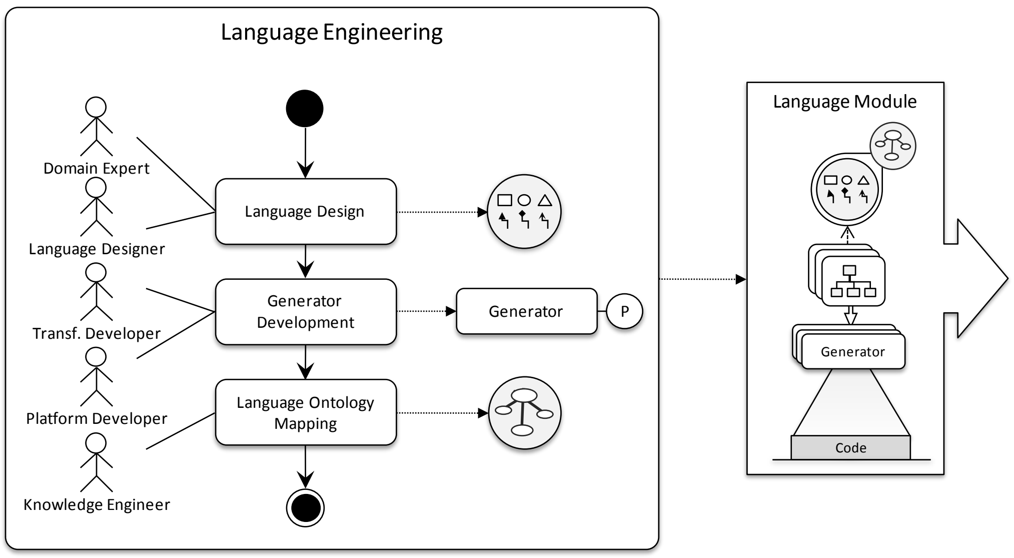


Figure 6.24: *Language engineering in HybridMDSD*

As depicted, domain expert and language designer do joint work during *language design*. For *generator development*, transformation and platform developer collaborate. The knowledge engineer is responsible to create an appropriate *language ontology mapping*, naturally also in corporation with the domain expert. The outcome of the language engineering phase is a language module, which is later used in aggregating projects. In the following, the enumerated activities are explained in detail.

6.3.3.1 Language Design

The language design phase is most vital in HybridMDSD development as it yields the central building blocks around mutli-domain engineering — the actual DSLs. It must be performed with care and foresight to ensure that once designed DSLs can be used as long as possible to leverage a maximal economic value. However, even the best-designed DSL becomes obsolete one day when new or changed requirements invalidate previously stated assumptions. Thus, it is crucial to properly scope a language's domain of discourse, its conceptual granularity, and technical level of abstraction to avoid undesirable domain overlaps, which would shorten the time until a language renders obsolete and results in redundant development. These preliminary steps are mainly accomplished during *project engineering* (see Section 6.3.4).

A typical DSL design process is subdivided into the phases (1) decision, (2) analysis, (3) design, (4) implementation, (5) deployment, and (6) maintenance [MHS05, Vis08, dH09]. In HybridMDSD language design we omit the phases (1), (4), (5), and (6). Phase (1) (the determination, whether it is appropriate to develop a DSL or not), (5) (when created DSLs are used), and (6) (when new requirements need to be implemented), is performed during project engineering and modeling, respectively (see Section 6.3.4 and 6.3.5). Phase (4) is implemented within generator development (see next section).

Phase (2) (domain analysis) comprises activities around "identifying and structuring information for reusability" [PDFG92, p. 6] in a particular domain. Typical sources are expert knowledge, existing code, technical documents, or customer surveys. In general, sources for domain analysis can be divided into the categories *informal*, *formal*, and *extract from code* [MHS05]. Several formal approaches exist, such as DARE [PDFG92], FAST [Wei95], or the previously introduced software product line enginneering method FODA (see Section 2.3). Nevertheless, domain analysis is often done informally, for example, based on existing implementations [Vis08]. Visser calls this approach "inductive" and highlights that it leads to quick results and ensures that DSLs can be implemented later on.

In HybridMDSD, the domain analysis phase is not necessarily to be performed in a formal, prescriptive way. Instead, we prefer a case-based approximation of a domain, i.e., the exploration of expressive domain concepts and relationships through the identification of prominent use cases (cf. Section 6.3.1.3). To this end, we too advocate the use of existing implementations to derive appropriate concepts and relationships. Reference implementations usually constitute proper realizations of the semantics of a domain. Hence, they help to comprehend processes and actors in that domain.

Figure 6.25 depicts the language design process in HybridMDSD, which starts with the *domain analysis* phase. As explained, this phase can be based on a reference implementation, which is usually created during generator development, introduced in the next section. In any case, the domain analysis phase yields several brief use case descriptions, which themselves may contribute to new parts of a reference implementation. Moreover, these use cases serve as basis for reference models that are later used to validate established language constructs. Both, reference implementations and use case descriptions, are ideal means to narrow concepts and relations of a domain.

Another source of information for that purpose are domain ontologies, as also depicted in Figure 6.25. The value of ontologies for domain analysis has been witnessed in recent years [GPvS02, Hru05]. Almeida et al. even coin the notion of *Ontology-based Domain Engineering (ODE)* [dAFGD02], in this respect. The basic idea is to adopt the domain-specific terminology and relationships from existing domain ontologies. In HybridMDSD, we clearly see potential for the application of such approaches. Our upper ontology USMO is the backbone for language ontologies, which inherit and specialize foundational concepts and relationships. To this end, it is possible to establish a layered architecture, including core and domain ontologies, which improves reuse in the ontology TS. In case a new DSL must be designed, these domain ontologies can be used to assist developers during language design. Figure 6.26 depicts this layered approach, which is aligned to the ontology hierarchy introduced in Section 2.4.1.2.

Succeeding to domain analysis, gathered information is directly transformed into the abstract syntax, i.e., the metamodel, of the domain-specific language that is to be created. In an it-

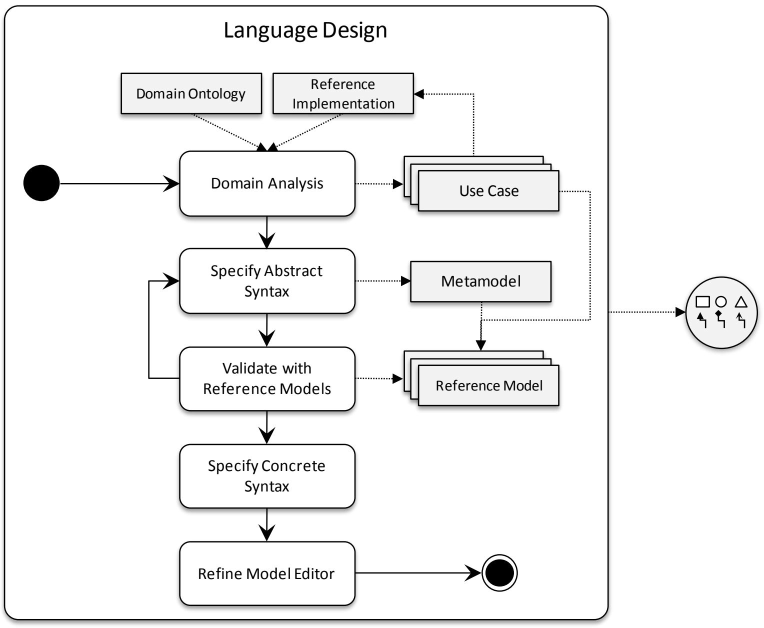


Figure 6.25: *Language design in HybridMDSD*

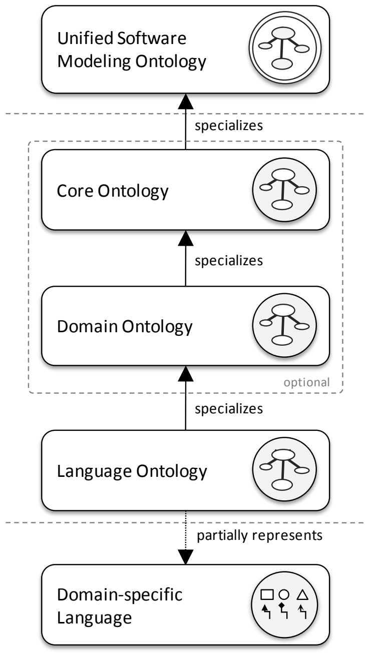


Figure 6.26: *Layered ontology architecture in HybridMDSD*

erative and incremental manner created structures are validated with use case-based reference models, which reveal conceptual and relational insufficiencies and lead to metamodel refinements. Once the metamodel renders stable and all the identified information from domain analysis is implemented, the DSL's concrete syntax can be specified. This phase may massively vary in development effort and resource consumption.

Depending on the type of notation that is appropriate for the developed DSL, it might be necessary to specify *textual*, *graphical*, or *hybrid* representations. Now, predominantly the modeling workbench that is used for language design has a significant effect on the development effort. Several modeling workbenches exist, as presented in Section 2.1.2.5. For instance, Kelly states that MetaCase's MetaEdit+ allows to develop graphical DSLs 2000 times faster compared to Eclipse EMF/GEF [Kel07]. However, more recently a new wave of tool suites for efficient development of textual modeling languages emerged [HJK+09, The09b], which are probably better suited for textual notations than MetaEdit+.

Language design in HybridMDSD is finally concluded with efforts to improve editing tools for model instances of the newly created DSL. This mainly comprises the creation of convenience features for the modeling process, such as syntax highlighting and code completion proposals in textual languages or dedicated layout constraints for graphical DSLs.

6.3.3.2 Generator Development

The language engineering phase of generator development comprises activities around the specification of generator templates to facilitate source code production from DSL instance models. Generator development usually starts with the development of a reference implementation [Bet04, SVEH07], as depicted in Figure 6.27.

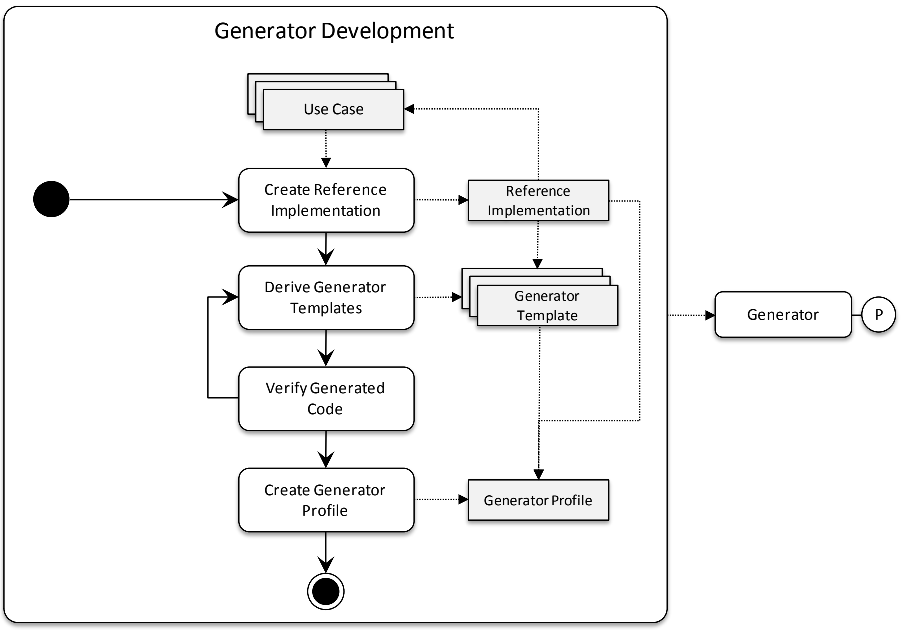


Figure 6.27: *Generator development in HybridMDSD*

To this end, use case descriptions that were formulated during language design provide an ideal starting point. A comprehensive reference implementation can then be utilized to derive programming patterns that are common to particular problem cases [Vis08]. These patterns are analyzed for variation points and transformed to generator templates, where the variation points are mapped to particular model queries based on the DSL metamodel. Later, when a dedicated DSM shall be generated, these queries are served with this model's content.

In principle, the goal in generator development is to result with generated code assets in code that is equal to what was previously specified as reference implementation. An ideal way to verify this is to create unit tests for the reference implementation and to apply these to generated code assets. When all reference implementation unit tests can be executed successfully with the generated code, generator development, at least from the perspective of the reference implementation, is complete. In order to test the code generator, appropriate reference models can be used that were created previously during language design (see section above).

The employment of use cases, reference models, and reference implementations in language design and generator development likewise, shows that the different development phases in language engineering are very close to each other. This highlights once more the importance of incrementatal and iterative development as all development phases interact tightly and depend on each other.

6.3.3.3 Language Ontology Mapping

The metamodel of a domain-specific language consists of concepts and relationships that are aligned to a particular domain. In contrast to an ontology for the same domain, which also contains concepts and relationships, the DSL is designed to facilitate the *prescriptive specification* of common affairs in that domain [AZW06]. Therefore, DSL modeling constructs are developed for the straightforward and convenient creation of model instances. Ontology engineering, instead, aims at representing a particular domain of discourse as precise as possible [Gui07]. An ontology shall facilitate the *descriptive representation* of affairs in a domain.

Hence, although DSLs represent the same domain of discourse like dedicated ontologies, they differ in choice and granularity of modeling constructs because of a different underlying intention[1]. To this end, the quality of a language with respect to the representational precision of a dedicated domain can even be measured with special properties, such as *soundness*, *completeness*, *lucidity*, and *laconicity* [Gui07].

Now, the purpose of the language ontology mapping phase in HybridMDSD is to (1) create an ontology that facilitates the representation of DSL models in terms of common concepts and relations from the software modeling domain, provided by USMO. Moreover, it aims at (2) the identification of a language's modeling constructs that *ontologically commit* to the created ontology (cf. Figure 6.26). To this end, two things need to be accomplished in this phase. First, the language ontology needs to be created and, second, appropriate mapping rules need to be specified that handle the instantiation of ontology individuals during modeling. In any case the starting point for development is the metamodel of the DSL to map, as depicted in Figure 6.28.

The first step is to identify the language's ontological commitment to the terminology proposed by USMO or specializing core or domain ontologies. Thus, we scope those modeling constructs that have a representation in terms of the corresponding ontology. This includes type, attribute, or reference definitions which are the most common modeling constructs in corresponding modeling languages. Once all constructs are identified, they are *lifted* from the modeling to the ontology technological space. The term "lifting", in this respect, was firstly coined by Kappel et al. who speak of "lifting metamodels to the semantic level" [KKK$^+$05, p. 2]. In the following, we explain the major lifting types.

Type Lifting denotes the creation of ontology classes for metamodel types or even type hierarchies. To this end, for every lifted type a corresponding ontology class is created in the language ontology. Inheritance relationships between lifted metamodel types can, if necessary, be easily reflected in the ontology TS.

Property Lifting denotes the creation of ontology entities for metamodel properties, including commonly available attributes and references. To this end, for every attribute or reference that shall be lifted, an ontology property or class is created. For example, a reference between two metamodel types could be lifted to an ontology concept that specializes USMO's RelationalProperty or Relator, depending on whether type or token models are involved.

[1] Of course, DSLs and ontologies also differ already because of the divers means to create corresponding terminology and relationships.

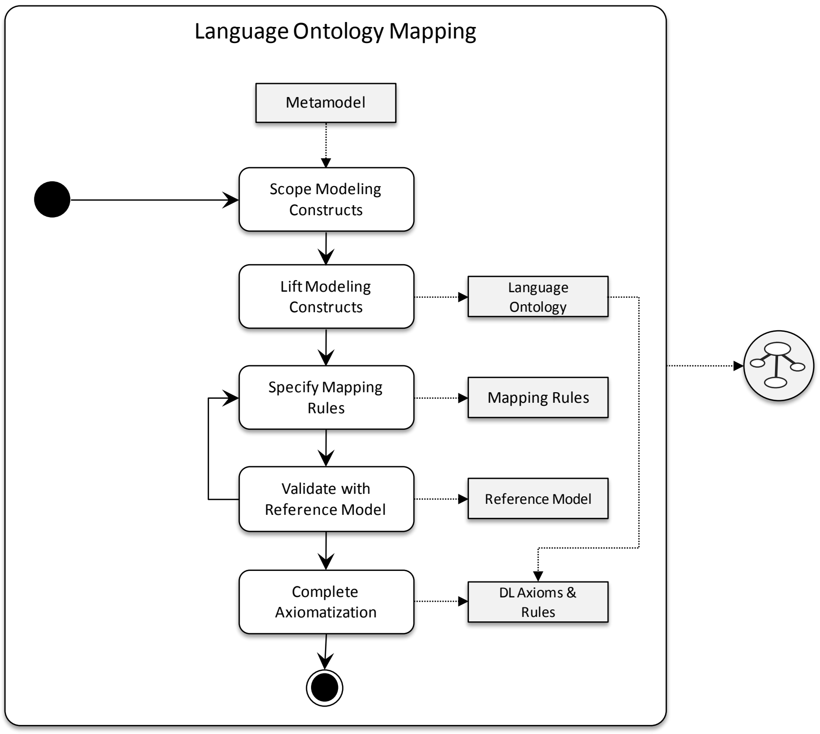


Figure 6.28: *Language ontology mapping in HybridMDSD*

Enumeration Lifting is a special type of lifting, which occurs only in modeling frameworks whose metametamodels include a corresponding concept. Enumerations are data types with a fixed value set. Therefore, corresponding types need to be lifted to an appropriate ontology class. This, in turn, is to be *closed* by providing an individual for each enumeration value and stating that the class is defined by exactly the set of these individuals. Such classes are then called *enumerated classes* [HKR$^+$04, p. 95].

Having all modeling constructs of the language's ontological commitment lifted, a first language ontology skeleton is ready. Now, before this skeleton can be completed with additional DL axioms and rules it must be validated, again, in an iterative and incremental process. To this end, appropriate mapping rules must be developed in the next step. These mapping rules control the instantiation of ontology constructs in coordination with available modeling constructs (cf. Figure 4.8). Once an initial set of rules is developed, reference models can be created and used for testing. These reference models could, for instance, be taken from those that were created during language design (cf. Section 6.3.3.1.

After several iterations both, ontological commitment and mapping rules should have been elaborated sufficiently. The last step during language ontology mapping is completion of the language ontology with appropriate DL axioms and rules. For example, to complete lifted inheritence structures it might be appropriate to ensure that an abstract super class that is extended by two subclasses is equipped with so-called *covering axioms* [HKR$^+$04, pp. 75-76]. For example, assumed are the ontology classes Car and its disjoint subclasses LuxuryCar and CompactCar.

$$\mathsf{LuxuryCar} \sqsubseteq \mathsf{Car}$$
$$\mathsf{CompactCar} \sqsubseteq \mathsf{Car}$$

Further assumed is that an individual necessarily needs to be either LuxuryCar or CompactCar. To specify this, the class Car needs to be *covered* by its subclasses, expressed by the following DL axioms.

$$\mathsf{Car} \equiv \mathsf{LuxuryCar} \sqcup \mathsf{CompactCar}$$

This concludes our explanation of the language engineering phase in HybridMDSD development. The next phase is the aggregated use of here developed language modules during project engineering, explained in the next section.

6.3.4 Project Engineering

In a reuse-centric environment like provided with HybridMDSD, the previously introduced language engineering phase does not need to be performed for each and every project ever and ever again. Instead, having a number of projects implemented, a critical mass of language modules should be available which eases project development massively. In any case, project development starts with the project engineering phase. Here, the modeling foundation for a software product or a product line is provided which is later (during modeling) employed to specify the actual modeling landscape. Figure 6.29 depicts the activities performed in this phase.

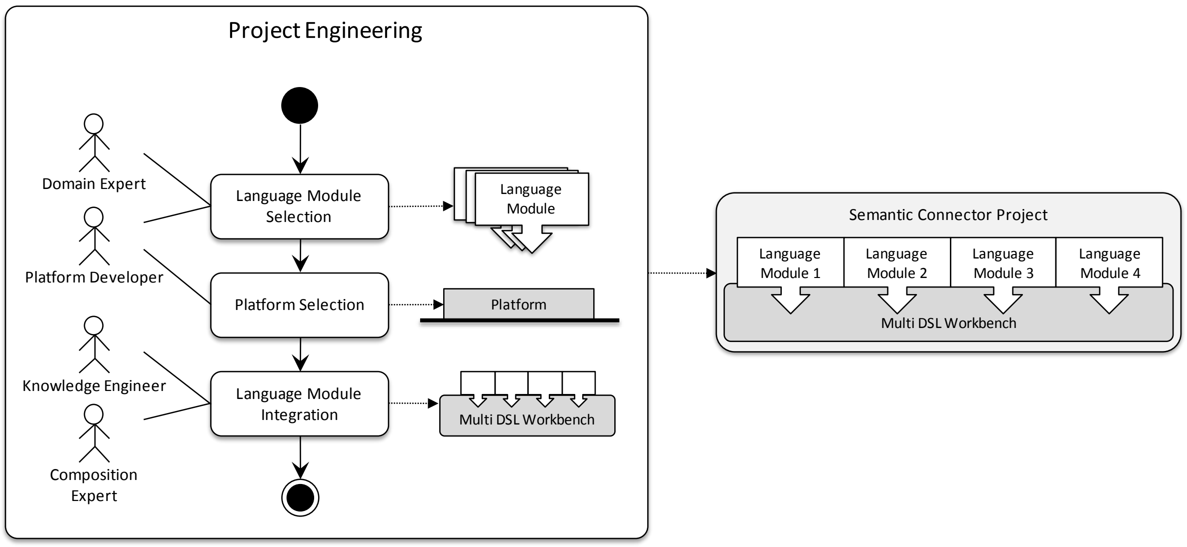


Figure 6.29: *Project engineering in HybridMDSD*

Firstly, appropriate language modules need to be selected from repository or created, if not available (see Section 6.3.4.1). Secondly, a particular runtime platform is chosen and selected language modules need to be adjusted accordingly (see Section 6.3.4.2). Lastly, all language modules that now provide necessary platform implementations need to be integrated on both, semantic and code level (see Section 6.3.4.3). The final result is a semantic connector project with a sophisticated multi-DSL workbench ready for modeling.

6.3.4.1 Language Module Selection

Recall from Section 6.3.3.1, that we omitted central steps in language design and referred to the project engineering phase. This included phase (1) decision for or against DSL development, (5) deployment of created DSLs and their active use, and (6) the maintenance of DSLs in case new requirements emerge. Figure 6.30 depicts how this is to be understood. In general, the language selection phase comprises activities around requirement estimation and verification whether identified requirements can be fulfilled by existing language modules. If this is not the case, the development process proceeds with language engineering to complete insufficient or to create new language modules. In this respect, language module selection comprises (1) decision, (5) deployment, and (6) maintenance.

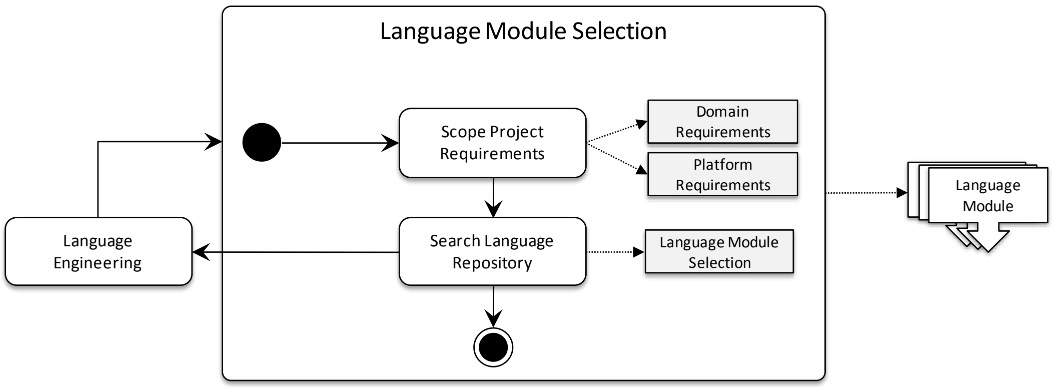

Language Module Selection
Scope Project Requirements
Domain Requirements
Platform Requirements
Search Language Repository
Language Module Selection
Language Engineering
Language Module

Figure 6.30: *Selecting appropriate languages modules*

The first task during language module selection is the determination of appropriate domain and platform requirements for the project to be developed. In this respect, the project's problem domain is analyzed and together with predefined product capability requirements, necessary platform features are approached. All together facilitates an estimation of necessary number, type, and complexity of the DSLs for the whole project. For example, a text console-based application which shall act in background does not require any DSL for user interface specification. Instead, a session-based internet game has high demands on UI capabilities but copes without persistent data structures.

Once a profound analysis has been performed, the language repository can be queried for necessary domain-specific languages. To this end, the comprehensive semantic information that is contained in language ontologies can be used to perform sophisticated queries. In an appropriate user interface, for instance, the top-level concepts of USMO could be used to provide a preliminary classification of different kinds of DSLs. This would lead to a general distinction between type, token, structure, behavior, and user interface languages. Subsequently, further concept-based queries could be posed to the language repository, quickly leading to desired results.

At the end of the language module selection phase, several language modules are chosen and can be integrated into a HybridMDSD modeling workbench for further tasks.

6.3.4.2 Platform Selection

In this development phase a particular product platform for the targeted project needs to be elaborated. Here, previously identified requirements and chosen language modules are involved. On the one hand, the platform requirements dictate particular demands such as the underlying operating system, certain security constraints, multi-user accessibility, or response times. On the other hand, the language modules already provide one or more platform generators that produce particular GPL code based on certain runtime environments and domain-specific frameworks.

Figure 6.31 depicts the one and only step in this phase is the validation of given platform requirements, i.e., the alignment of predefined demands with provided capabilities. In case the given language module selection does not fulfill stated requirements, another language engineering iteration must be performed. The final outcome of this phase is the *project platform* in the sense of the definition given in Section 2.1.3.1.

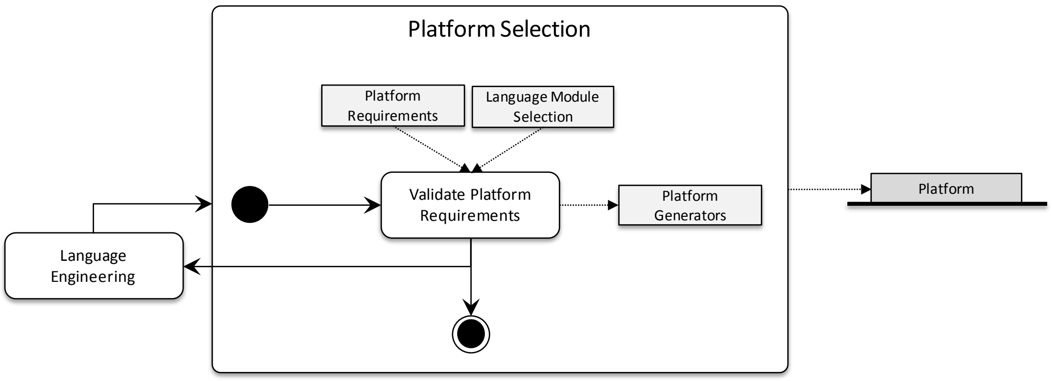


Figure 6.31: *Selecting a project platform*

6.3.4.3 Language Module Integration

The last phase before the actual model creation can begin is the integration of selected language modules. This comprises both, the integration on a semantic level, i.e., the combination of different language ontologies, and the integration on code level, i.e., the definition of code composition structures between assets that will be generated from domain-specific models of employed language modules. Figure 6.32 depicts the activites involved in this development phase.

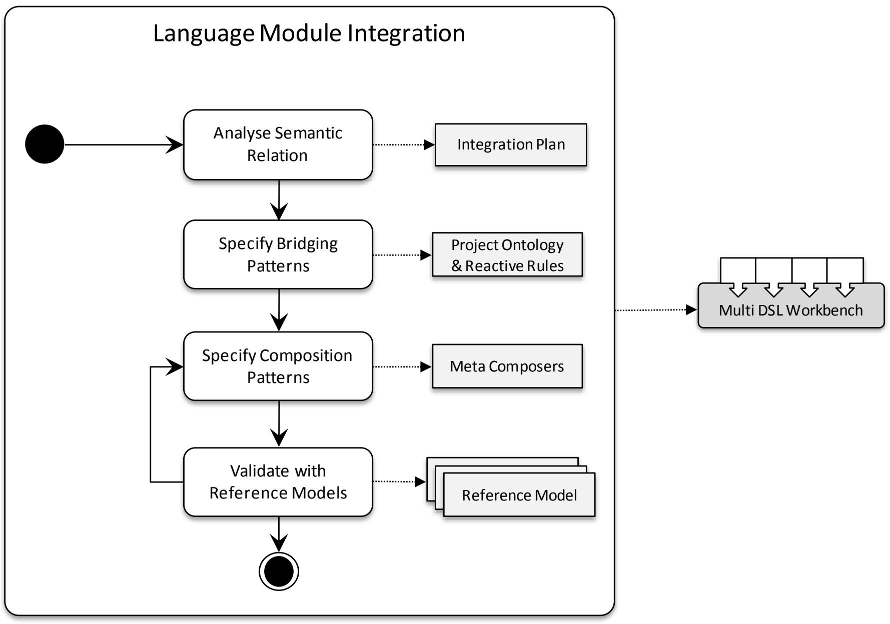
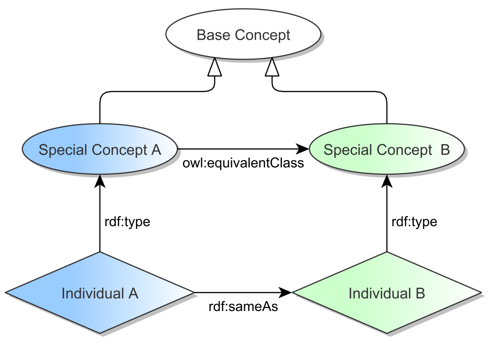

Language Module Integration
Analyse Semantic Relation
Integration Plan
Specify Bridging Patterns
Project Ontology & Reactive Rules
Specify Composition Patterns
Meta Composers
Validate with Reference Models
Reference Model
Multi DSL Workbench

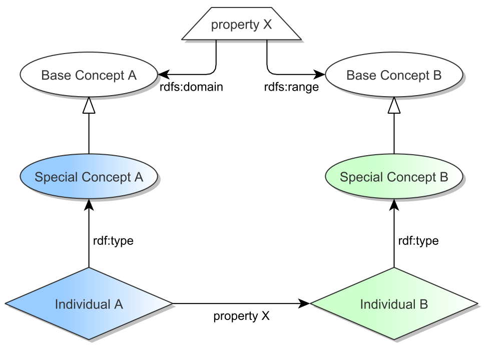
Figure 6.32: *Integrate language modules*

The phase starts with an analysis of the semantic relation between different language modules, i.e., the language ontologies of these modules. To this end, we identified three possible types of relation, as depicted in Figure 6.33. In all parts of the figure, colored entities belong to language ontologies and non-colored entities to our upper ontology. To this end, all individuals are aggregated in an overall project konwledgebase.

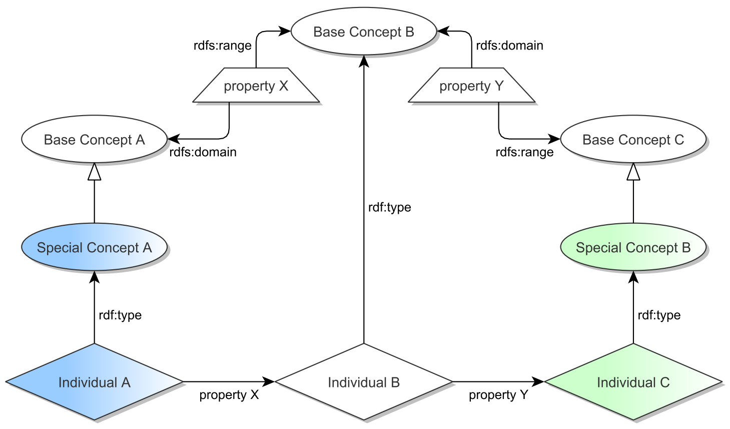
Semantic Equivalence is the most basic form of relation between two languages and exists if particular modeling constructs are equivalent to each other. In this respect, corresponding language ontology concepts would have to be related through the OWL properties



(a) Equivalence between DSLs



(b) Simple relation between DSLs



(c) Complex relation between DSLs

Figure 6.33: *Different relations between DSL modeling constructs*

owl:equivalentClass and owl:sameAs to denote that two concepts or individuals are the same (cf. Figure 6.33 (a)). Unfortunately, owl:equivalentClass only expresses that the extensions of two classes are the same and not that two concepts are semantically equal. This would be possible with an application of owl:sameAs for ontology classes, which is only allowed in the (undecidable) OWL Full standard [BvHH^{+}04]. However, a DL reasoner would infer that an individual of type A would be of type B, if A and B are related via `owl:equivalentClass`, which suffices for us at the moment.

Simple Relation between two DSLs exists in case concepts in adjacent language ontologies are directly related through predefined properties from the central upper ontology USMO. For example, if one language contains modeling constructs that map to the Schema concept and another language contains modeling constructs that map to the Substantial concept, then both modeling constructs are inherently related through the USMO property is_instance_of. Figure 6.33 (b) illustrates the general constellation of this relation type.

Complex Relation is a kind of relation that involves one or more concepts and relations to bridge two DSLs. Thus, modeling constructs in adjecent DSLs are not only related through equivalence or simple properties but are rather indirectly linked through a number of concepts and individuals, respectively. Figure 6.33 (c) shows an exemplary case where another foundational concept from the upper ontology is instantiated to bridge two specialized language ontology concepts. A form of complex relation arises, for instance, when combining behavioral languages. Recall from Section 6.1.2.2, that Actions are triggered by Events. Now, assumed a pageflow language allows the definition of transitions and an action DSL is empoyed to define invokable functionality, then a bridging concept — the Event — is needed that facilitates the binding of both modeling constructs and does not belong to one of the integrated DSLs.

As outcome of the semantic relation analysis an *integration plan* is formulated, which contains a list of all concept pairs that need further elaboration for integration. Here, we clearly state that our comprehensive upper ontology, including concepts, relationships, and axiomatization, provides an optimal basis which makes a detailed elaboration of language relations often superfluous. However, like presented in the following particular inter-language relations need a designated, self-explanatory identifier to facilitate convenient use during modeling. Thus, at least the name of particular relations needs to be adjusted.

The elaboration of particular inter-language connections is summarized with the *specify bridging patterns* task in Figure 6.32. Here, integrating properties that subclass appropriate upper ontology properties and reactive rules (see Section 2.4.3.2) that manage the instantiation of necessary bridging individuals for complex relations, are developed. Both are part of the overall *project ontology*. Moreover, this ontology imports all employed language ontologies and is the basis for instantiation during the modeling phase (see below).

The last constructive part of the language module integration phase before validation is the specification of composition patterns for language integration on code level. Recall from Section 6.2, that this involves the creation of *composition generators* for particular inter-language connections. To this end, every inter-language connection that has a representation on code level, needs to be equipped with an appropriate composition generator. The necessary implementation effort depends massively on the kind of code generated from the language modules to integrate. For instance, very heterogeneous assets might necessitate sophisticated integration platforms, such as CORBA or RMI. Thus, the composition generator would have to produce appropriate interface definitions which generated assets would have to be adapted in a way that they implement these interfaces. In contrast, homogenous generated assets that use one single platform, such as Java or C++, can be integrated already with simple invasions using means of object-orientation or parameterization.

After having created all necessary composition generators, language module integration also on code level is complete. Like all the other phases of HybridMDSD development, we again use reference models to test whether the development result is insufficient or error-prone. Here, corresponding reference models are not anymore restricted to one single language module like in previous phases, but rather involve DSLs of several modules. Additionally, references between these models need to be established to validate developed integration patterns.

6.3.5 Modeling

The previous sections presented how different languages are mapped to the ontology technological space and how they are integrated into project-specific scenarios. The language engineering phase covered tasks around DSL design, generator development, and language ontology mapping (cf. Section 6.3.3). The project engineering phase covered language module selection, platform elaboration, and the creation of integrating structures on semantic and programming code level (cf. Section 6.3.4). Both, language engineering and project engineering are premises for the actual modeling phase, i.e., software engineering using abstract DSLs. During this phase, it's time to reap the fruits of previously invested efforts.

At this point, it is important to elucidate that the preliminary tasks must be performed *only once*. The reuse-centric repository architecture of HybridMDSD allows to reutilize various types of development assets, including DSLs, ontology mappings, generators, and editors. Also, integrational assets such as inter-language connections and corresponding composition generators can be organized in the repository. Hence, after implementing a number of language modules and specifying appropriate integration structures, new software projects can be easily started without any preparatory work. This massively reduces the amount of financial efforts for the implementation of new projects and, in addition, shortens the rampup phase for new developers who can start to contribute their domain-specific expertise from day one.

Figure 6.34 depicts the activities during the modeling phase. As illustrated, in this phase we distinguish between *product* and *product line* engineering. Thus, development might start with the consideration of the problem space or directly with the creation of solution space models.

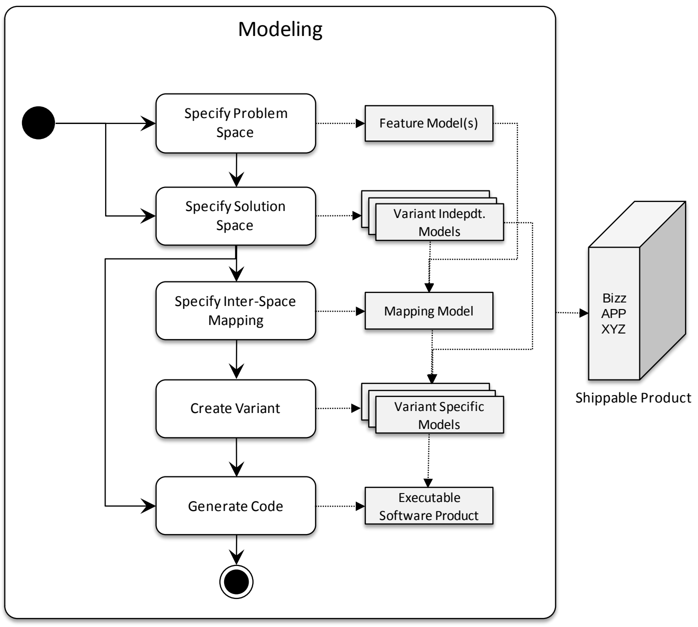


Figure 6.34: *Modeling in HybridMDSD*

6.3.5.1 Problem Space Modeling

During problem space modeling domain experts elaborate, usually together with a product managment team, one or more feature models that specify the variability of the product line to build. This commonly includes variability in customer-communicable product functionality. Moreover, additional variability can be defined for technical features, such as platform switches or the ability to react on changes in an application's runtime context [Fea07]. Recall from Section 2.3, that the development of feature models can be performed with well established methods, such as FODA.

6.3.5.2 Solution Space Modeling

When specifying the solution space, previously captured domain and project requirements (see Section 6.3.4) as well as the functionality described in feature models are used as functional specifications that instruct modelers in the development process. In case a comprehensive product line is developed, solution space modeling results in *variant independent models (VIMs)* [HL06], i.e., models that contain solution space parts for a set of product variants.

Variant independent models are not necessarily consistent because they might contain the implementation of contradictory features. For instance, if two alternative features affect the data type of an entity's attribute then the corresponding VIM would contain two equally named attributes, each with a different data type. In case the underlying DSL demands uniquely named attributes the VIM would interfere with the constraints of its defining language and, thus, would be inconsistent.

The development of solution space models is a *sequential* task, no matter if VIMs in a product line or usual models that do not contain variability are created. Thus, because of the dependencies between different models, inconsistencies in the solution space are common during model creation. User interface models shall display entities that are defined with data structure languages, page flow languages link dialogs that are specified with UI DSLs, and so on and so forth. In this respect, the HybridMDSD architecture alleviates the challenges that arise in various ways.

Preserving Consistency

First, inconsistencies between arbitrary models of different DSLs can be automatically detected and presented to the modeler. The semantic architecture of HybridMDSD facilitates the definition of consistency constraints on different levels. Thus, description logics axioms can be used to define sufficient and necessary conditions for konwledgebase individuals. More complex cases may involve rule-based integrity constraints. For details and illustrative examples the reader is referred to Chapter 7.

Decreasing Modeling Efforts

Second, the employment of reactive rules allows to define actions that are to be carried out when certain conditions are met (cf. Section 2.4.3.2). This way, developers can define templates that supersede trivial modeling steps. An example is the automatic enforcement of *declare-use* relationships between different languages. For better understanding we consider the following brief example.

Assumed are two DSLs, one to define data structures and another one that facilitates user interface specification. Further assumed is that the UI DSL allows to define named data types that may be bound to content widgets, such as tables or lists. The corresponding metamodel class is only a shallow stub without any additional properties besides its name attribute. The deeply specified data structures are defined in the data DSL. The corresponding semantics is that the UI language *uses* the data type definitions that are declared in the data DSL.

Now, if the modeler starts modeling with the creation of the user interface and declares a data type that is directly bound to a content widget, then the actual *definition* of this data type is missing. In this case, it is obvious that a corresponding definition must be provided with the data DSL. A reactive rule could implement the creation of appropriate konwledgebase individuals that had to be “reflected” into the modeling technological space. Another possibility is to trigger corresponding model transformations that implement this task. This transformation, in turn, could have been previously created using similarity analysis based on employed language ontologies. Again, the reader is referred to Chapter 7 for a detailed presentation of these features.

Guiding the Modeling Process

Last, the semantic infrastructure of HybridMDSD serves as ideal basis for further guidance capabilities. The application of deductive rules facilitates to provide further modeling assistance for developers. This already begins with the creation of inter-model references. When model elements in adjacent models of different DSLs shall be linked, the underlying project ontology is queried for available relations. This is done by posing a query that asks for all relations, which have the `rdfs:domain` property set to the ontology class that maps to the selected model element. Similarly, all available individuals that suffice the class definition defined by the `rdfs:range` property can be queried and presented to the user as valid reference peers.

Moreover, rule-based deductions facilitate the employment of *domain-specific* adaptation rules. This allows for both, the application of dedicated guidance recipes during the creation of new models and the application of particular repair recipes in case of inconsistencies caused by incorrect choices in existing models. For example, in particular workflow and dialog languages it might be desireable that resulting system models do not contain any gaps in workflows or navigation paths. To this end, appropriate rules can be defined that (1) identify and classify such gaps on the basis of the semantic infrastructure and (2) fix identified gaps if desired by the modeler. Chapter 7 gives insight also to this kind of use case.

6.3.5.3 Mapping Specification

In order to control variant-specific variability in a software product line, problem space feature models need to be mapped to solution space artifacts. This is done with the mapping specification task. Chosen variants, i.e., feature subsets of specified feature models, are the basis to derive a solution space subset adequate for a dedicated product instance. There are various ways to define the mapping between features and solution space artifacts.

IFDEF Statements

The probably most trivial way to implement feature dependencies is the employment of if-like statements directly in implementing programming code. In this respect, a very prominent example are C++ IFDEF statements that are evaluated by a preprocessor before the actual compiler starts working. Depending on the evaluation result, the preprocessor includes or excludes marked code snippets.

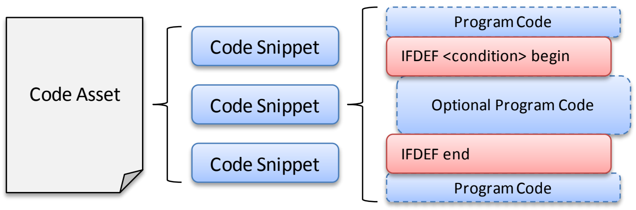


Figure 6.35: *Variability implementation with IFDEF statements*

Figure 6.35 illustrates the IFDEF concept. Thus, code assets are composed of several snippets written in a particular programing language. Selected snippets can be encapsulated with preceding and following IFDEF statements. Here, the first statement usually contains the condition that is evaluated by the precompiler.

Piece Lists

Another rather trivial implementation of the variability problem is the management of "piece lists", i.e., sets of countable artifacts that contribute to the solution space. This usually comprises class files, resource files, and other types of artifacts. The prominent variability management tool *pure::variants* [pur06] exploits this concept with a maximum of flexibility. The software enables developers to define their own metamodel for the employed solution space. This abstract variability-centric representation of the solution space is then called *family model* [pur09, p. 21]. Corresponding family model elements are mapped to appropriate counterparts in the solution space. Common family model elements are, for example, directories, files, or java classes. During variant instantiation the family model — which actually denotes one of the above mentioned piece list — is restricted according to the features that are included in the chosen variant.

Figure 6.36 illustrates how the piece list concept is implemented in pure::variants. The figure

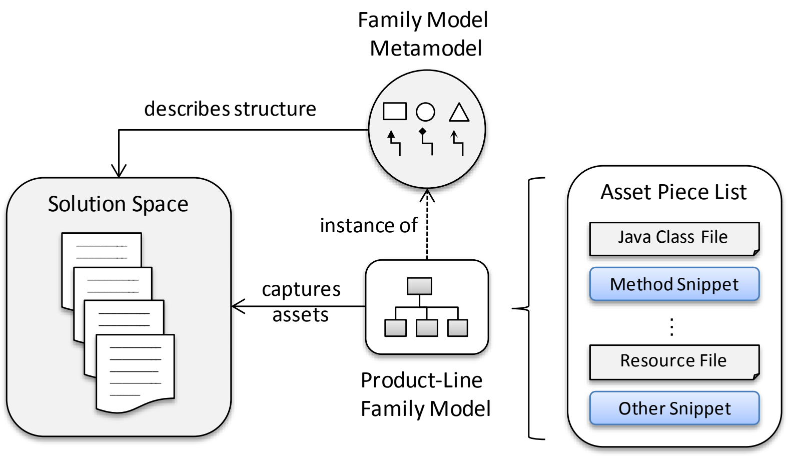


Figure 6.36: *Variability implementation with pure::variants piece lists*

depicts how the solution space is captured with family models. The actual restriction process based on a particular feature selection is not shown.

Comprehensive Feature Mapping

Because HybridMDSD considers product line development in *model-driven* software applications, more sophisticated approaches to manage variability in the solution space are necessary. Above sketched approaches were developed for countable source code artifacts and can not easily be applied to comprehensive specification models, because these only denote abstract representations of subsequently generated code artifacts. An interesting approach to solve this problem was introduced by Heidenreich et al. with the *FeatureMapper* [HKW08]. This tool allows to define a direct binding between features of a feature model and particular parts of solution space models, such as type instances, attributes, or property values. Corresponding mappings denote transformation operations that are to be executed during variant instantiation. These operations include entity removal or addition and property value changes.

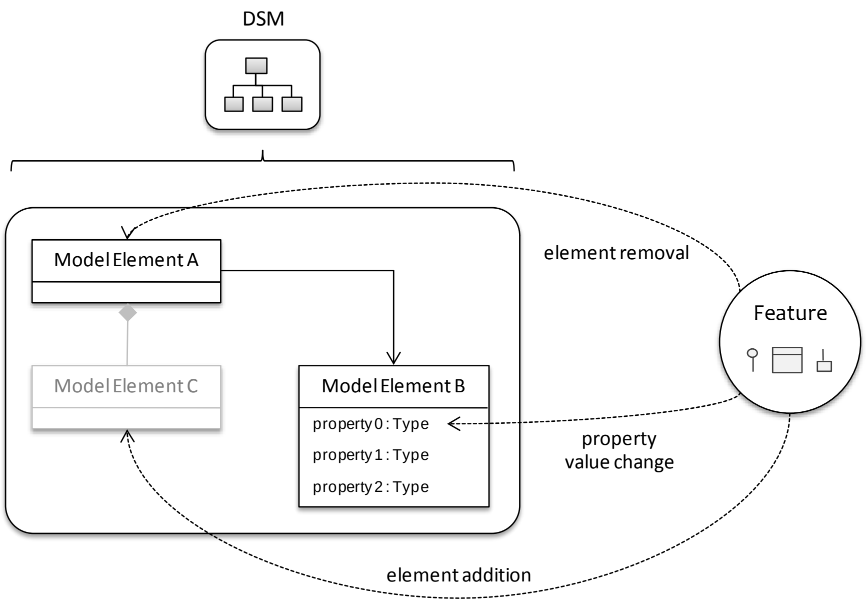


Figure 6.37: *Variability implementation with the FeatureMapper*

Figure 6.37 illustrates how the FeatureMapper works. The figure depicts a domain-specific model that contains several elements. Moreover, a particular feature is shown that maps to dedicated model elements by specifying certain mapping transformation operations on the model's contents.

Dependency Resolution

Either way, the creation of appropriate mapping specifications is a complex task. The solution space of a model-driven software product line comprises many different domain-specific models, each contributing a functional excerpt to the final application. Hence, a consumer-communicable abstract product feature might be manifested in various models at a time, as illustrated in Figure 6.38.

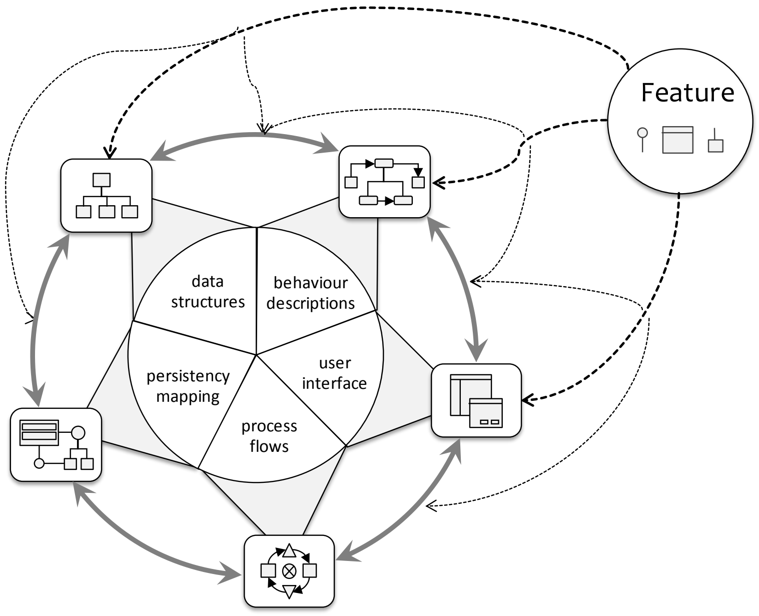


Figure 6.38: *Complex feature implementations in a multi-DSL product line*

Thus, when creating a mapping specification, all locations in the solution space that belong to the feature that shall be mapped must be identified. This is difficult, time-consuming, and error-prone if no further assistance can be provided for the developer.

With HybridMDSD mapping features to solution space artifacts is massively simplified. The connections between domain-specific models are captured in a foundational project konwledgebase. Hence, if a feature is mapped to a particular model element, the knowledgebase can be queried for dependencies to adjacent models. This way, the developer is always aware about the dependencies that must be additionally considered.

Chapter 7

Validation

This chapter comments on the validation of the previously developed conceptual results. To this end, we implemented a comprehensive framework and an IDE integration prototype that realizes the HybridMDSD architecture to proof general feasibility (see Section 7.1). We used the prototype to implement a comprehensive industrial-scale model-driven software product line in context of the german and the european research projects feasiPLe [Fea07] and AMPLE [Con07a] (see Section 7.2). Moreover, we illustrate the application of outstanding HybridMDSD features (in Section 7.3) and discuss the results (in Section 7.4).

7.1 Prototypical Implementation

7.1.1 Overview

We developed a prototypical implementation of the HybridMDSD approach by (1) working out a framework to integrate the involved technological spaces (see Section 7.1.2) and (2) embedding this framework into the plugin architecture of the Eclipse IDE (see Section 7.1.3). In summary, this resulted in 7 Eclipse plugin projects for the core implementation and 4 plugin projects that provide enabling tools or slightly different versions of existing plugins.

Developed plugins contribute several Eclipse-specific extensions to the IDE. Besides minor contributions such as problem markers and context menus, this includes 2 project natures, 2 project builders, 2 wizards, 7 views, and 3 perspectives. In total the prototype counts 22.883 lines of code. Table 7.1 illustrates statistics about the plugin infrastructure.

7.1.2 A Framework to Integrate Technological Spaces

The first and foremost step towards an implementation of the HybridMDSD approach was to develop a framework that provides input, modification, and output operations for the assets that shall be processed. Recall from previous chapters that the HybridMDSD solution approach unifies two technological spaces, namely the modeling and the ontology technological space. Hence, it was necessary to (1) develop a framework for the processing of ontologies and modeling entities (see next section) and (2) design a concept to bridge entities of both types (see Section 7.1.2.2).

Plugin	Lines of Code	Contributions
	core plugins	
com.sap.research.hmdsd.core	8705	2 Project Natures, 2 Project Builders
com.sap.research.hmdsd.core.ui	5137	5 Views, 3 Perspectives, 2 New Project Wizards
com.sap.research.hmdsd.model.ecore	2469	Ecore Adapter for Modeling Technological Space
com.sap.research.hmdsd.ontology.jena	4604	Jena Adapter for Ontology Technological Space
com.sap.research.hmdsd.composition	372	Composition Generator for oAW-based Meta Composers
com.sap.research.hmdsd.composition.ui	840	2 Views
com.sap.research.hmdsd.logging	756	Logging Capabilities
	enabling plugins	
com.sap.research.hmdsd.feature	*not applicable*	Feature Deployment
com.sap.research.hmdsd.updatesite		Update Site Deployment
org.apache.commons.lang		Apache Commons Lang Export
org.openarchitectureware.emf.generic.editor		oAW Generic Editor Adaptation
in total: 22.883		

Table 7.1: *Code statistics of the HybridMDSD prototype*

7.1.2.1 Pivot Modeling

Before introducing the pivot models we developed to represent relevant entities for HybridMDSD, it is necessary to highlight one important design decision. The communities of both TSs that need to be processed already provide APIs and frameworks to manage corresponding resources. For ontology processing there are, e.g., Jena [Jen09] or Sesame [Adu09], for metamodeling appropriate frameworks are provided with the corresponding modeling technology, such as Eclipse EMF [Ecl]. Thus, we were confronted with the question whether yet another API should be designed rather than using existing ones.

To make a long story short, we decided to go the former way and designed our own framework. We did this mainly for the following two reasons:

1. Concerning the modeling TS, the decision was rather obvious. Like analyzed in Section 4.1.1, HybridMDSD is not restricted to one single modeling technology. But, the limitation of the applied modeling API to one single, existing framework would prevent or at least hamper the inclusion of other technologies. Additionally, to not end up in the same dilemma with our own modeling API, it was necessary to restrict the entities in this API to the lowest common denominator between available technologies, with respect to commonly employed modeling constructs.

2. Concerning the ontology TS the reason to employ a self-designed framework was not justified with the variety of available technologies but rather with theoretical considerations. Until now, we assumed an application of description logics, rule languages, appropriate serialization formats like OWL and RDFS, and Semantic Web Reasoners for the implementation of our semantic infrastructure. However, there are also alternative approaches to model ontologies such as the the object-oriented ontology language *F-Logic (Frame Logic)* [KLW95]. In contrast to the open world assumption of the Semantic Web, F-Logic follows a closed world view and integrates logical rules without any additional languages. A self-designed ontology API was crucial to leave to choice for available options open.

Container Infrastructure

To obtain a maximum of flexibility while maintaining a minimum of redundancy when implementing a memory model for desired entities, we analyzed involved technological spaces for structural similarities. This way, we derived a common pivotal core model that specializing implementations for both spaces could leverage. In principle, the structure of both technological spaces is very similar. Both share a certain *container infrastructure* where, simply spoken, resources contain other resources. Such containers hold a unique identifier and contained resources are addressed relatively to this identifier. Thus, ontologies are identified with XML namespace URIs [SWM04], which usually also applies for metamodels. Figure 7.1 depicts the core interfaces of the container API.

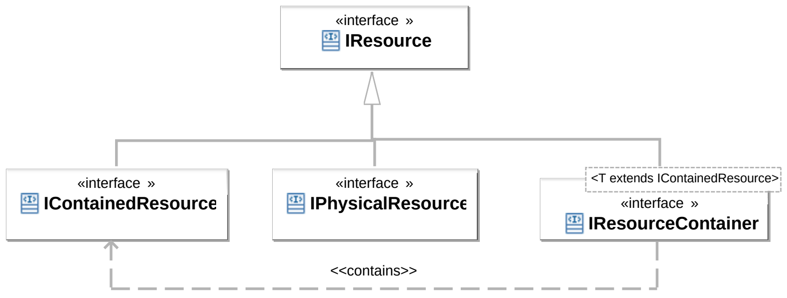


Figure 7.1: *HybridMDSD container infrastructure API*

The figure shows the IResource interface, which is the base interface for all in-memory representations of involved assets. An IResource provides methods to obtain the unique identifier and the relative name of an ontology or a modeling resource. Specializing interfaces declare the container infrastructure with the interfaces IResourceContainer and IContainedResource. The interface IPhysicalResource provides persistence support to common resources by adding serialization and deserialization operations.

Modeling Technological Space

Now, as indicated above the container infrastructure is the basis for further technological space-specific extensions. The first extension, in this respect, was developed to process modeling entities. To this end, the container infrastructure API was extended with modeling-specific interfaces and base implementations. Here, we proceeded straightforward by analyzing constructs that are common to the most modeling frameworks, like introduced above. The resulting API for metamodeling is depicted in Figure 7.2.

The interface IMetamodel inherits from IResourceContainer and contains IMetamodelElements, which are IContainedResources. This is properly implemented through a restriction on the inherited type parameter T to appropriate IMetamodelElement specializations. The metamodel elements that we have identified as common denominator for type modeling are IClass and IProperty — the latter with the child classes IAttribute and IReference. Additionally, to represent enumerations we employ the interfaces IEnumeration and IEnumerationValue.

As a side notice, we need to state that the modeling tool that we have employed to produce depicted diagrams, Omondo UML [Omo09], was not able to depict parameterized type parameters. Thus, the type parameter of the interface IProperty is not (as depicted) of the type Object but rather restricts the type parameter to the type IClass<?,?>. The same applies for the type parameters RangeType and DeclaringType of the interfaces IAttribute and IReference, respectively. Unfortunately, this error traverses also the following diagrams.

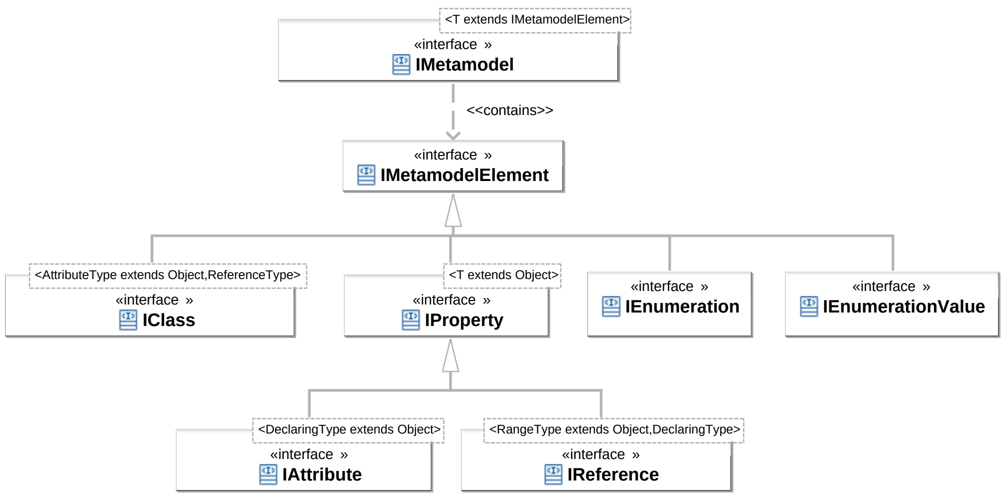


Figure 7.2: *HybridMDSD metamodeling API*

Ontology Technological Space

As within the modeling API, we created appropriate interfaces and base classes for the ontology technological space. Equally straightforward we analyzed the structure of ontologies and modeled most common entities with the top-level interfaces depicted in Figure 7.3.

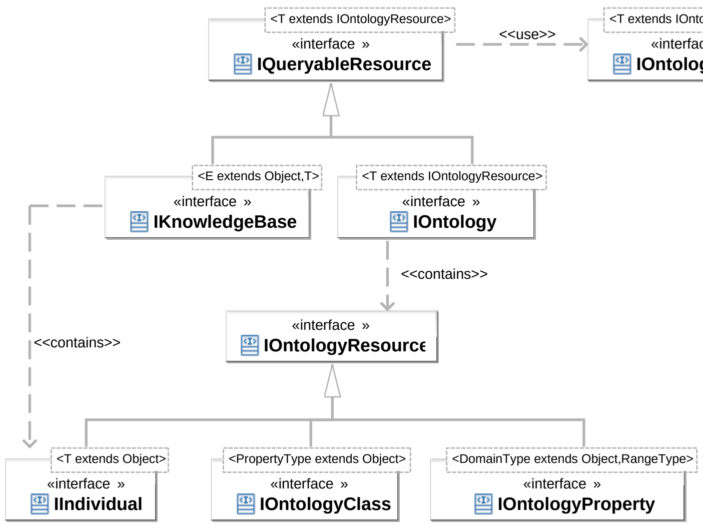


Figure 7.3: *HybridMDSD ontology API*

The Semantic Web community provides sophisticated query languages to facilitate the usage of potentially very complex knowledge structures. To this end, available languages include both, approaches comming from official standardization efforts, like *SPARQL (SPARQL Protocol and RDF Query Language)* [PS08] or RDQL [Sea04], and project-specific approaches, like the SPARQL extension ARQ provided by the Jena Semantic Web framework [Jen09].

To enable query processing for ontologies in our HybridMDSD prototype, the root interface for ontology containers is the interface IQueryableResource, which in turn inherits from IResourceContainer and restricts contained resources to IOntologyResources. IQueryableResource provides query capabilities on the basis of IOntologyQuery instances. These in turn, are used to hold platform-specific query strings that are processed by corresponding ontology API implementations.

Recall from Section 2.4.1.1 that we clearly distinguish between TBox and ABox containers. In the ontology API this is reflected in the two container interfaces IOntology and IKnowledgeBase. Consequently, IOntologys might only contain IOntologyClass and IOntologyProperty instances, while IKnowledgeBases are allowed to contain only IIndividuals.

Implemented Specializations

As already indicated in the code statistics table (see Table 7.1), we implemented one platform-specific implementation for each pivot model. In this respect, we provided an *Ecore adapter* for the modeling TS and a *Jena adapter* for the ontology TS. Corresponding pivot model specializations restrict the core interface layer with appropriate type parameter restrictions and adapt available framework implementations.

7.1.2.2 Modeling Ontology-Space Bridge

After having briefly introduced the framework for processing modeling and ontology resources, this sections sheds light on the binding between both technological spaces. This binding mainly comprises two distinct parts. On the one hand, we have developed a dedicated tool ontology that provides important relationships to establish a binding between modeling entities and ontology individuals. On the other hand, we designed a particular framework that observes the modeling landscape and handles knowledgebase instantiation.

Semantic Connector Utility Ontology

The semantic core of HybridMDSD constists of (among others) the upper ontology USMO that provides necessary concepts to represent important ingredients of domain-specific languages and models in a multi-DSL software engineering workbench. However, the bare ontology classes and properties from USMO do not allow to *map* actual ontology individuals to particular modeling entities. Yet, this is necessary to establish a partial semantic representation of the modeling landscape as it was conceived in the previous chapters. As a solution we developed a light-weight utility ontology that contributes the necessary concepts and relationships. Figure 7.4 depicts the result.

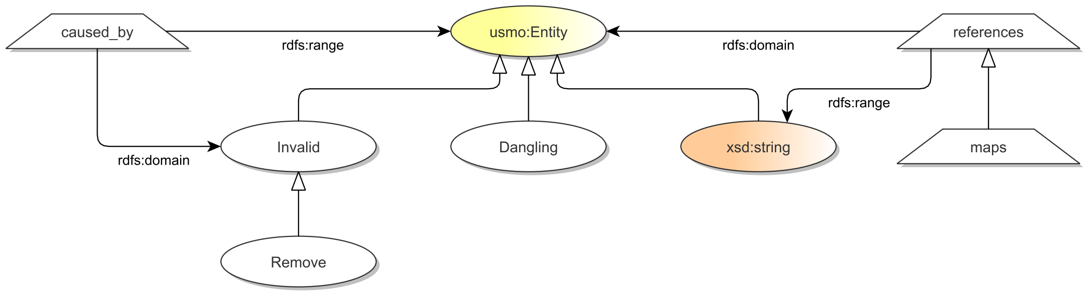


Figure 7.4: *The semantic connector utility ontology*

The most important entities provided by the *semantic connector* ontology (indicated with the prefix `sc`) are the two properties maps and references. These properties are essential to connect ontology individuals with entities from the modeling technological space. To this end, the OWL data type properties are assigned with string values that identify such entities with a unique identifier. This identifier is platform-specific, depending on the employed modeling technology. In Eclipse EMF, for example, such identifiers are special URIs employing the dedicated `platform` protocol. Corresponding URIs have then the form `platform:/<project name>/<project-relative path>/<resource>`.

Employing two distinct properties to reference modeling entities is justified with non-trivial mappings between the integrated TSs. Thus, a modeling construct might map to more than one single ontology class depending on its semantics. For instance, recall from Section 6.1.2.2 that a user interface button denotes a Representative and an Event likewise. To maintain the inter-space mapping it is important to mark every knowledgebase individual with information that indicates what modeling entity caused its creation.

To this end, we distinguish between *loose* and *tight* references. Loose references are captured with the references property and record any individual that was created for a particular modeling construct. In contrast, tight references capture the *most intentional* individual a modeling entity maps to. For the semantics of the button example above, the tight reference would map the corresponding Representative individual rather than the Event individual. This is because a UI button denotes above all a widget in a dialog, no matter which events it may produce. Tight references are expressed with the maps property, which is subsumed by references and may point to one single entity only.

Figuring out which semantics is primary and which secondary is up to the knowledge engineer and the language designer, who specify the mapping in a joint venture during language engineering (cf. Section 6.3.3).

Space Bridge Architecture

The semantic connector ontology is a necessary utility to establish a connection between the knowledgebase of a particular project and the corresponding modeling landscape. The last missing step towards a complete integration of involved technological spaces is an appropriate architecture that provides the machinery to execute language module-specific mapping rules. Figure 7.5 illustrates this architecture.

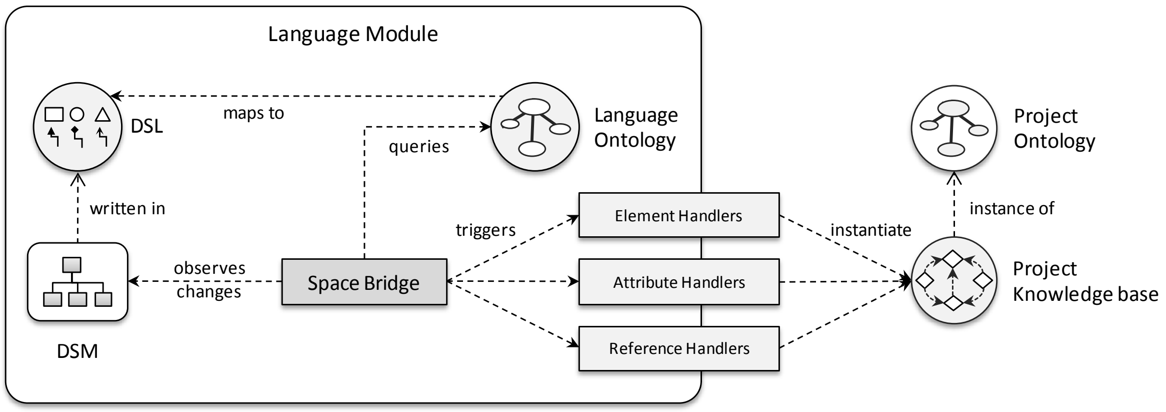


Figure 7.5: *Bridging technological spaces in HybridMDSD*

The figure shows a language module that provides a DSL and a language ontology, as introduced above. Because the domain-specific models that are created in the solution space instantiate exactly one DSL metamodel coming from a language module, the figure also shows the corresponding DSM in the module's context. Now, the new mapping-specific components are the depicted Space Bridge and associated change handlers. Each language module must provide an appropriate Space Bridge implementation that aggregates suitable handler instances. For both, the HybridMDSD core framework offers particular interfaces and base classes.

Concerning the change handlers, we distinguish between the three possible change classes that might occur in the modeling landscape. These are (1) instance creation or removal, (2) the change of an attribute value, and (3) the change of a reference value. For each change class, dedicated interfaces and base classes are provided that differ in available change handling methods. Thus, the interface `ITypedInstanceHandler` for instance creation and removal provides the methods `handleAdd(InstanceType instance)` and `handleRemove(URI instanceUri)`, while the interface `ITypedAttributeHandler` only declares the method `handleChange(InstanceType instance, AttributeType attribute)`.

Each change handler returns the name of the metamodel construct it manages. Thus, this might be the unique identifier of a metamodel class, an attribute, or a reference. After a modification in an observed model, the core space bridge implementation, which is encapsulated in the type `AbstractTypedSpaceBridge` and each language module space bridge inherits from, automatically resolves matching handlers for (1) the affected modeling construct and (2) the type of change that occurred. Additionally, simple one-to-one projections from modeling constructs to ontology concepts are conveniently instantiated by the framework. To this end, corresponding ontology concepts can be equipped with predefined comment values, which are processed by the core space bridge. Further mapping logic is then provided by dedicated handler classes. Here, non-trivial mapping operations can be implemented.

7.1.3 Eclipse Workbench Integration

The Eclipse IDE is an extensible open source tools project which is based on the flexible plugin concept of the OSGi Service Platform [All09]. In short, the strategic goals of Eclipse according to the most recent version of its roadmap [Con08] are:

1. Establish Eclipse as a leading provider of open source runtime technologies.
2. Maintain global leadership in open source tools platforms.
3. Create value for all its membership classes.
4. Foster growth of the ecosystem, particularly in verticals.

One step to reach these goals was made already from the very beginning of Eclipse development with the so-called *extension registry* [CR06, pp. 112-113]. The extension registry allows plugin vendors to provide properly defined *extension points* for their implementations. These extension points are used by third party vendors who deliver proper implementations that extend the existing platform. Although the extension concept overlaps with the ideas of the OSGi Service Platform and there are also discussions on the advantages and disadvantages of both [Bar07], it is still one major component of the Eclipse plugin infrastructure.

One of the core components of the Eclipse IDE is the *Plug-in Development Environment (PDE)* [The09a]. This component provides sophisticated tools and APIs for plugin development on the basis of Eclipse. In this respect, a number of extensions points are declared that facilitate the enrichement of the default workbench with additional layouts (or perspectives), tool-specific views, different types of wizards, project-specific builders, and so-called project natures. In the following we introduce the extensions we developed for the integration of our HybridMDSD prototype.

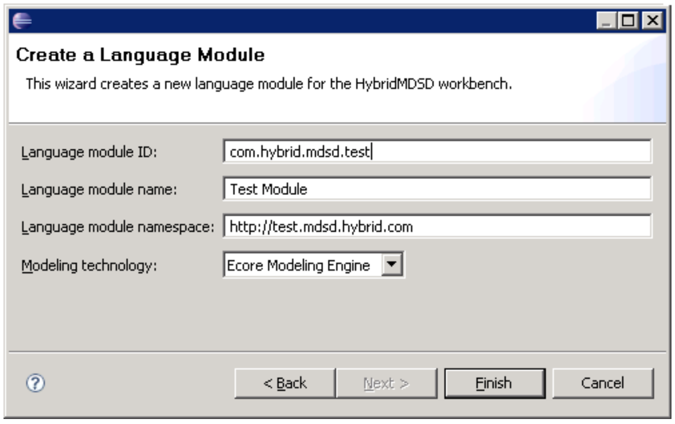
7.1.3.1 Natures

A project in Eclipse is first and foremost a particular file system resource with a small set of meta information that contains, e.g., the project name. On top of such trivial data, the respective type of a project is determined by so-called *project natures*. A project might have multiple natures at a time while each nature adds additional metadata and further processing instructions. A Java project, for instance, is associated with the corresponding Java project nature that adds information about the build path, compiler compliance settings, or particular code style templates.

In HybridMDSD, we have defined two project natures. Recall from Section 4.2.2 and 6.3, that we distinguish development in HybridMDSD into language engineering and project engineering. Corresponding assets that result from these development phases are reusable language modules and integrating semantic connector projects, respectively. For each type of asset we provide a particular project nature.

Language Module Nature

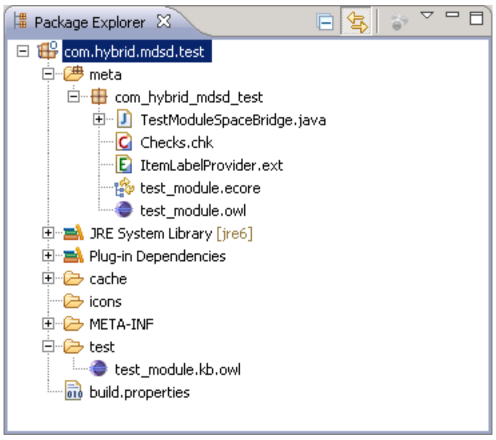
The language module nature enables developers to create a project that contains a metamodel, corresponding code generators, tools and editors, a language ontology, and a mapping specification in form of a space bridge implemetation with appropriate handlers. Each language module is bound to a certain modeling technology and its spacebridge uses modeling stack-specifc types accordingly. For the creation of language module projects, we developed a convenient wizard that is illustrated in Figure 7.6.


Create a Language Module
This wizard creates a new language module for the HybridMDSD workbench.

Language module ID: com.hybrid.mdsd.test
Language module name: Test Module
Language module namespace: http://test.mdsd.hybrid.com
Modeling technology: Ecore Modeling Engine

Figure 7.6: *The language module creation wizard*

As depicted, the developer is asked for a unique identifier, a human-readable name, and an XML namespace for each new language module. In this respect, the wizard conveniently derives the namespace from the entered identifier. The combo box in the bottom of the dialog lets the developer select a target modeling technology for the new module. As stated above, this box currently only contains the ecore modeling engine.



```
com.hybrid.mdsd.test
  meta
    com_hybrid_mdsd_test
      TestModuleSpaceBridge.java
      Checks.chk
      ItemLabelProvider.ext
      test_module.ecore
      test_module.owl
  JRE System Library [jre6]
  Plug-in Dependencies
  cache
  icons
  META-INF
  test
    test_module.kb.owl
  build.properties
```

Figure 7.7: *Language module project structure*

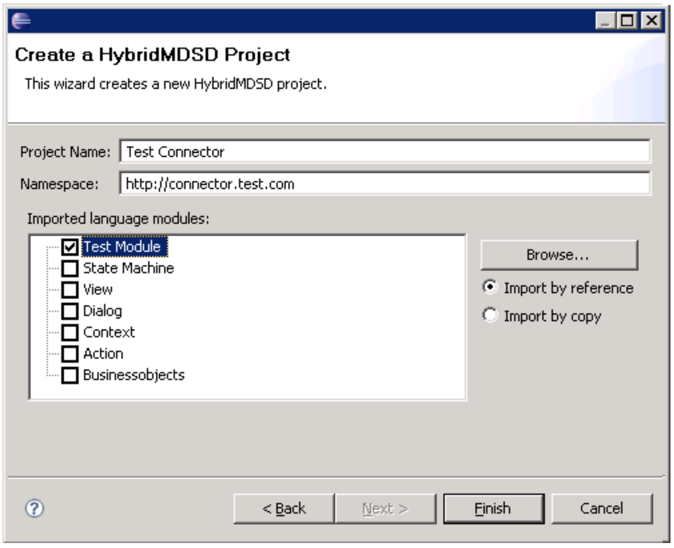
Having all necessary input provided the wizard creates the initial project structure illustrated in Figure 7.7. The most important folders in the new language module project are “meta” and “test”.

The meta folder contains different files for the module’s metamodel (“test_module.ecore”), its language ontology (“test_module.owl”), and a space bridge implementation (“TestModuleSpaceBridge.java”). Moreover, we implemented the creation of openArchitecture-based model editor enhancement files (“Checks.chk” and “ItemLabelProvider.ext”). These files allow to elaborate modeling editors with domain-specific tree node renderings and completion proposals.

The test folder represents the testing ground for the language module developer. The HybridMDSD framework observes this folder for changes and invokes the space bridge implementation immediately when new models are created or existing models are modified in this folder. This enables straightforward verification of implemented mappings. In addition, the test folder contains the knowelege base file (“test_module.kb.owl”) the HybridMDSD framework associates with the space bridge for testing. Hence, this is a sort of tryout project knowledgebase.

Semantic Connector Nature

Projects with the semantic connector nature are prepared to import and aggregate previously defined language modules. A semantic connector project defines a project ontology where all employed language ontologies are imported and dedicated bridging properties can be created on demand. Like the language module nature, also the semantic connector nature comes with a project creation wizard, depicted in Figure 7.8.


Create a HybridMDSD Project

This wizard creates a new HybridMDSD project.

Project Name: Test Connector

Namespace: http://connector.test.com

Imported language modules:

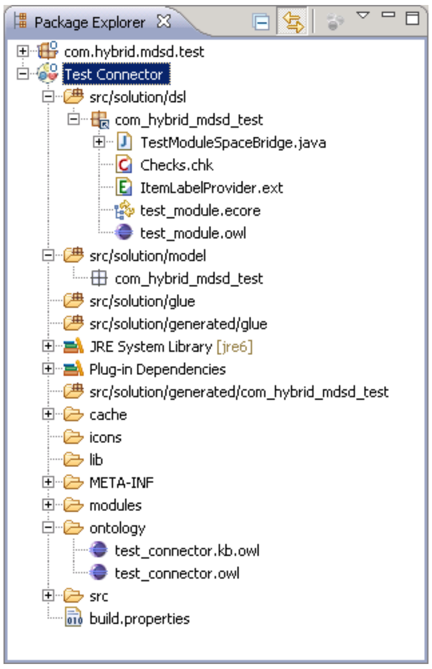
- [✓] Test Module
- [ ] State Machine
- [ ] View
- [ ] Dialog
- [ ] Context
- [ ] Action
- [ ] Businessobjects

[✓ Import by reference] Import by copy

Figure 7.8: *The semantic connector creation wizard*

A semantic connector project has a name and a namespace. We omitted a unique identifier for semantic connector projects because they are not subject for reuse and, hence, must not be addressable in a shared repository. The bottom area of the connector creation wizard lists available language modules, the connector project shall import. The wizard lists all language modules that the current Eclipse workbench contains. Additional modules can be loaded from the file system. Hence, it becomes obvious that the implementation of the conceived language module repository is restricted to file system resources in our prototype.

The semantic connector wizard creates a project structure like shown in Figure 7.9. The project structure employs several source folders (all below the root source folder “src”) to properly distinguish diverse development areas. Thus, the src folder contains a “solution” and a "generated" folder. As the name indicates, the solution folder contains assets from the solution space. In case a product line is created, the optional "problem" folder is available as well.



```
com.hybrid.mdsd.test
Test Connector
  src/solution/dsl
    com_hybrid_mdsd_test
      TestModuleSpaceBridge.java
      Checks.chk
      ItemLabelProvider.ext
      test_module.ecore
      test_module.owl
  src/solution/model
    com_hybrid_mdsd_test
  src/solution/glue
  src/solution/generated/glue
  JRE System Library [jre6]
  Plug-in Dependencies
  src/solution/generated/com_hybrid_mdsd_test
  cache
  icons
  lib
  META-INF
  modules
  ontology
    test_connector.kb.owl
    test_connector.owl
  src
  build.properties
```

Figure 7.9: *Sermantic connector project structure*

Following the layered stack we illustrated in previous sections (cf. Figure 4.6, 6.20, and 6.21) we divided the solution folder into the subfolders "dsl" that contains M2 modeling assets, "model" that contains M1 modeling assets, "glue" that contains composition generators, and "generated" that contains both, generated code from each language module and generated gluing code.

Below each subfolder, particular folders are located that are dedicated to corresponding language modules. For example, the illustrated test project that imports only the above shown "Test Module" contains folders named by the module's unique identifier. Here, the distinction between *import by reference* and *import by copy* elucidates. Referential import links the meta folder of an imported language module directly into the integrating connector project, while the copy import replicates necessary contents.

7.1.3.2 Builders

Also the build process for projects in the Eclipse IDE is organized with project natures. Thus, a nature might be associated with one or more *project builders*, resolved via unique identifiers. The corresponding extension points in PDE allow to define particular builder implementations for specified identifiers. The overall Eclipse build process, which is entirely controlled by the platform framework, becomes aware about involved builders in case a certain nature is assigned to a project and calls these builders respectively. This way, project resources are automatically processed by different builders during the build process.

For our HybridMDSD prototype we developed two builders, one for each of the above introduced project natures. The HybridMDSD project builders are based on the Eclipse builder framework that triggers project builders in case project content has changed. Hence, each time a model was modified and saved, a corresponding change event is captured by the builder framework that causes the invocation of associated builders.

Language Module Builder

In brief, the language module builder implements the space bridge architecture depicted in Figure 7.5. Thus, the builder observes all models that instantiate the language module's metamodel. To this end, a dedicated folder that contains these models is watched for changes (cf. above). In case a relevant resource (i.e., an associated model) was modified, the language module builder informs the corresponding modeling framework to calculate the changes between the current model and the model that was cached during the preceeding save action. Necessary model caching operations are performed by the language module builder as well.

The underlying modeling framework, which is concerned with the model difference calculation, succeedingly fires model change events that are caught by the language module's space bridge. In this respect, the language module builder also realizes that the space bridge is properly registered to newly created and unregistered from old deleted models. As described in Section 7.1.2.2, the space bridge then implements the invocation of individual modeling construct handlers, which instantiate the knowlege base.

Semantic Connector Builder

Like a semantic connector project integrates several language modules, also the semantic connector builder integrates corresponding language module builders. Hence, the functionality of this builder implementation is actually limited to the invocation of all builders from imported language modules when relevant project resources change.

7.1.3.3 Perspectives

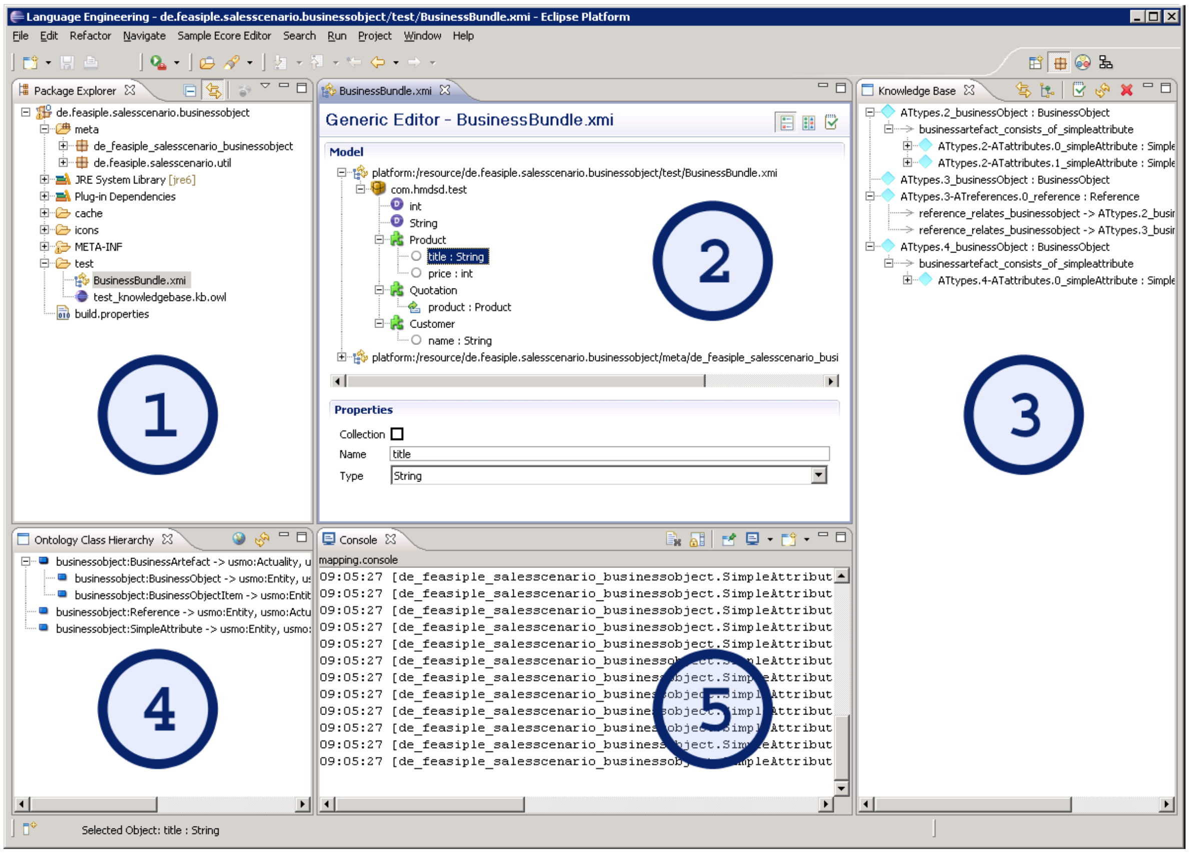
Eclipse perspectives are layout presets that arrange editors and views that are relevant for a particular context. For example, the out of the box Java perspective arranges in one Eclipse workbench the so-called *package explorer* (as depicted in Figure 7.9 and 7.7), an appropriate Java code editor, and an outline view that illustrates types, fields, and methods of the edited Java class file. In the HybridMDSD prototype, we created three different perspectives, one for each major development phase (see Section 6.3.2.2).

Language Engineering

The first perspective is dedicated to the language engineering phase in the HybridMDSD development process. A screenshot of the perspective is shown in Figure 7.10. The language engineering perspective is indicated by the package symbol shown in the perspective selection menu (top-right corner). The symbol is also used as icon for language module projects, as illustrated in the *package explorer* (area (1)) that contains the language module `de.feasiple.salesscenario.businessobject`.

Area (2) shows the *generic editor* provided by openArchitectureWare as an extended version of the default EMF reflective editor. The default reflective editor renders EMF models in a tree widget, where containment relationships between model elements are respresented as parent child relationships in the tree. The editor directly (reflectively) processes a model's metamodel to render attribute setup widgets and to provide model completion proposals. As already mentioned in Section 7.1.3.1, the generic editor allows to override the default tree node rendering with particular oAW resources. This way, tree node labels and icons can be presented appropriately tailored to a certain domain.

While the areas (1) and (2) show third party views, area (3) depicts the first self-defined HybridMDSD view. The area shows the *knowledgebase view* that gives insight to the actually selected project's knowledgebase. A detailed introduction is given in the next section. Area (4)


Language Engineering - de.feasiple.salesscenario.businessobject/test/BusinessBundle.xmi - Eclipse Platform
File Edit Refactor Navigate Sample Ecore Editor Search Run Project Window Help
Package Explorer
de.feasiple.salesscenario.businessobject
meta
de_feasiple_salesscenario_businessobject
de.feasiple.salesscenario.util
JRE System Library [jre6]
Plug-in Dependencies
cache
icons
META-INF
test
BusinessBundle.xmi
test_knowledgebase.kb.owl
build.properties
BusinessBundle.xmi
Generic Editor - BusinessBundle.xmi
Model
platform:/resource/de.feasiple.salesscenario.businessobject/test/BusinessBundle.xmi
com.hmdsd.test
int
String
Product
title : String
price : int
Quotation
product : Product
Customer
name : String
platform:/resource/de.feasiple.salesscenario.businessobject/meta/de_feasiple_salesscenario_busi
Properties
Collection
Name
title
Type
String
Knowledge Base
ATtypes.2_businessObject : BusinessObject
businessartefact_consists_of_simpleattribute
ATtypes.2-ATattributes.0_simpleAttribute : Simple
ATtypes.2-ATattributes.1_simpleAttribute : Simple
ATtypes.3_businessObject : BusinessObject
ATtypes.3-ATreferences.0_reference : Reference
reference_relates_businessobject -> ATtypes.2_busir
reference_relates_businessobject -> ATtypes.3_busir
ATtypes.4_businessObject : BusinessObject
businessartefact_consists_of_simpleattribute
ATtypes.4-ATattributes.0_simpleAttribute : Simple
Ontology Class Hierarchy
businessobject:BusinessArtefact -> usmo:Actuality, u
businessobject:BusinessObject -> usmo:Entity, us
businessobject:BusinessObjectItem -> usmo:Entit
businessobject:Reference -> usmo:Entity, usmo:Actu
businessobject:SimpleAttribute -> usmo:Entity, usmo:
Console
mapping.console
Selected Object: title : String
1
2
3
4
5

```
09:05:27 [de_feasiple_salesscenario_businessobject.SimpleAttribut
09:05:27 [de_feasiple_salesscenario_businessobject.SimpleAttribut
09:05:27 [de_feasiple_salesscenario_businessobject.SimpleAttribut
09:05:27 [de_feasiple_salesscenario_businessobject.SimpleAttribut
09:05:27 [de_feasiple_salesscenario_businessobject.SimpleAttribut
09:05:27 [de_feasiple_salesscenario_businessobject.SimpleAttribut
09:05:27 [de_feasiple_salesscenario_businessobject.SimpleAttribut
09:05:27 [de_feasiple_salesscenario_businessobject.SimpleAttribut
09:05:27 [de_feasiple_salesscenario_businessobject.SimpleAttribut
09:05:27 [de_feasiple_salesscenario_businessobject.SimpleAttribut
09:05:27 [de_feasiple_salesscenario_businessobject.SimpleAttribut
09:05:27 [de_feasiple_salesscenario_businessobject.SimpleAttribut
```

Figure 7.10: *The language engineering perspective*

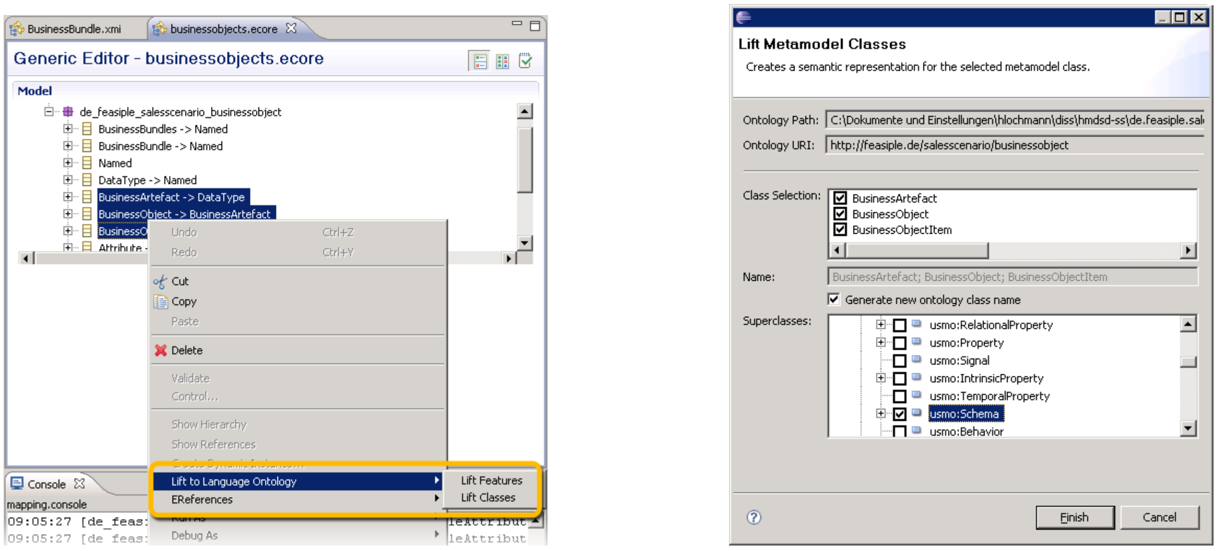
shows another HybridMDSD view, the *ontology class hierarchy* view. As the name indicates, this view renders the structure of ontologies depending on the selected project. Again, the next section gives a detailed introduction.

Area (5) in Figure 7.10 shows the *mapping console*. As the implementation of mappings between modeling and ontology TS can become very complex, it is crucial to assist mapping developers with debugging tools. To this end, the HybridMDSD framework allows to place logger calls directly in the implementation of space bridges and modeling construct handlers. Corresponding logging output is directly forwarded to the mapping console.

In summary, the language engineering perspective arranges all necessary views to conveniently develop language modules. The package explorer gives insight about module resources. The generic editor allows to create test models for iterative and use case-driven development of mapping specifications. The knowledgebase view renders created individuals after each modeling step. Together with the mapping console, this facilitates fast and efficient development of mapping specifications.

Finally, what we did not consider until now is the creation of the language ontology, which is another important task besides the mapping specification during language engineering. To this end, the implementation of a full-blown ontology editor was by far out of scope in this work. Therefore, we advocate the employment of well-known external tools for ontology design, such as Protégé. Nevertheless, we decided to assist language module developers with additional tooling to create at least a language ontology skeleton from existing metamodels.

Therefore, we provided small utility that allows developers to select particular metamodel constructs and to lift them to language ontologies, as described in Section 6.3.3.3. This is done



Creates a semantic representation for the selected metamodel class.

Ontology Path: C:\Dokumente und Einstellungen\hlochmann\diss\hmdsd-ss\de.feasiple.sal
Ontology URI: http://feasiple.de/salesscenario/businessobject
Class Selection: ☑ BusinessArtefact ☑ BusinessObject ☑ BusinessObjectItem
Name: BusinessArtefact; BusinessObject; BusinessObjectItem
☑ Generate new ontology class name
Superclasses: ☐ usmo:RelationalProperty ☐ usmo:Property ☐ usmo:Signal ☐ usmo:IntrinsicProperty ☐ usmo:TemporalProperty ☑ usmo:Schema ☐ usmo:Behavior

(a) Lifting context menu (b) Lifting dialog

Figure 7.11: *Lifting metamodel structures to language ontologies*

through a context menu item, as illustrated in Figure 7.11 (a). In case of type lifting, the utility creates ontology classes for the selected metamodel types. To this end, a lifting dialog facilitates the selection of ontology base classes for newly created classes, as depicted in Figure 7.11 (b). Here, an appropriate tree widget renders the type structure of the central upper ontology USMO.

Project Engineering

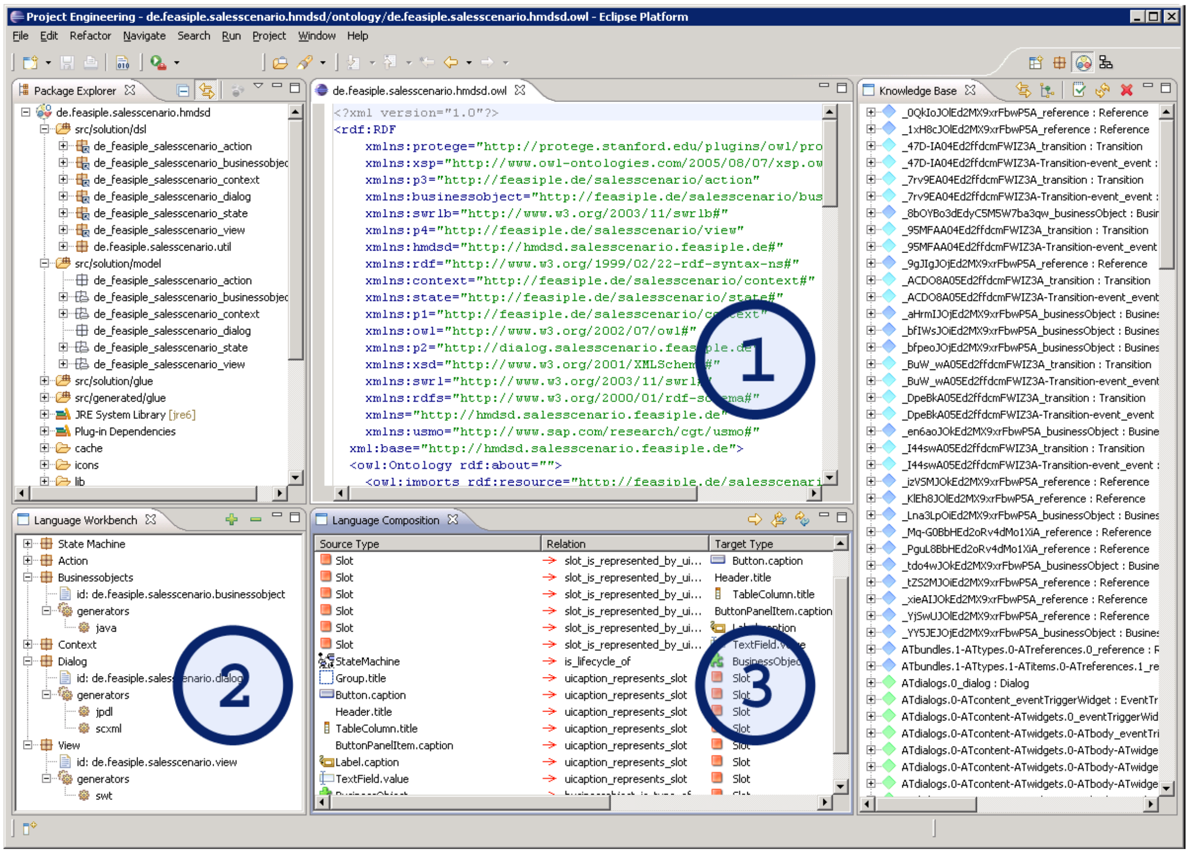
Succeeding to language engineering, HybridMDSD development proceeds with the project engineering phase. The corresponding perspective is shown in Figure 7.12. Both the package explorer and the knowledgebase view are already known from the language engineering perspective. In contrast to the previous perspective, the package explorer does not contain language modules but rather semantic connector projects, as these projects result from project engineering.

Area (1) in Figure 7.12 highlights a primitive XML-based ontology editor that we have created with minimal effort to provide at least direct editing support for ontology resources. Area (2) shows the more interesting *language workbench view*. This view gives insight about imported language modules. In area (3), the *language composition view* is marked. Here, developers are able to maintain inter-language connections and corresponding composition generators. Both language workbench and language composition view are explained in detail in the next section.

In summary, the project engineering perspective arranges views and editors that facilitate the intergation of various language modules into a semantic connector project. The language ontologies of all integrated modules are automatically imported into the project ontology. Again, with the help of external tools inter-language connections can be specified in case existing relationships derived from the upper ontology are not sufficient. Each inter-language connection is depicted in the language composition view, where the developer is able to create particular code composition patterns with dedicated composition generators.

Modeling

The third HybridMDSD perspective was designed to assist the actual modeling phase in multi-DSL development. Figure 7.13 depicts a screenshot. Once more, the common package explorer and knowledgebase view are represented also in this perspective. Additionally, in area (1) of



Project Engineering - de.feasiple.salesscenario.hmdsd/ontology/de.feasiple.salesscenario.hmdsd.owl - Eclipse Platform
File Edit Refactor Navigate Search Run Project Window Help
Package Explorer
de.feasiple.salesscenario.hmdsd.owl
Knowledge Base
Language Workbench
Language Composition
1
2
3

```
de.feasiple.salesscenario.hmdsd
  src/solution/dsl
    de_feasiple_salesscenario_action
    de_feasiple_salesscenario_businessobjec
    de_feasiple_salesscenario_context
    de_feasiple_salesscenario_dialog
    de_feasiple_salesscenario_state
    de_feasiple_salesscenario_view
    de.feasiple.salesscenario.util
  src/solution/model
    de_feasiple_salesscenario_action
    de_feasiple_salesscenario_businessobjec
    de_feasiple_salesscenario_context
    de_feasiple_salesscenario_dialog
    de_feasiple_salesscenario_state
    de_feasiple_salesscenario_view
  src/solution/glue
  src/generated/glue
  JRE System Library [jre6]
  Plug-in Dependencies
  cache
  icons
  lib
```

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:protege="http://protege.stanford.edu/plugins/owl/pro
    xmlns:xsp="http://www.owl-ontologies.com/2005/08/07/xsp.ow
    xmlns:p3="http://feasiple.de/salesscenario/action"
    xmlns:businessobject="http://feasiple.de/salesscenario/bus
    xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
    xmlns:p4="http://feasiple.de/salesscenario/view"
    xmlns:hmdsd="http://hmdsd.salesscenario.feasiple.de#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:context="http://feasiple.de/salesscenario/context#"
    xmlns:state="http://feasiple.de/salesscenario/state#"
    xmlns:p1="http://feasiple.de/salesscenario/context"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:p2="http://dialog.salesscenario.feasiple.de"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:swrl="http://www.w3.org/2003/11/swrl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns="http://hmdsd.salesscenario.feasiple.de"
    xmlns:usmo="http://www.sap.com/research/cgt/usmo#"
  xml:base="http://hmdsd.salesscenario.feasiple.de">
  <owl:Ontology rdf:about="">
    <owl:imports rdf:resource="http://feasiple.de/salesscenari
```

```
_0QkIoJOlEd2MX9xrFbwP5A_reference : Reference
_1xH8cJOlEd2MX9xrFbwP5A_reference : Reference
_47D-IA04Ed2ffdcmFWIZ3A_transition : Transition
_47D-IA04Ed2ffdcmFWIZ3A-Transition-event_event :
_7rv9EA04Ed2ffdcmFWIZ3A_transition : Transition
_7rv9EA04Ed2ffdcmFWIZ3A-Transition-event_event :
_8bOYBo3dEdyC5M5W7ba3qw_businessObject : Busir
_95MFAA04Ed2ffdcmFWIZ3A_transition : Transition
_95MFAA04Ed2ffdcmFWIZ3A-Transition-event_event
_9gJIgJOjEd2MX9xrFbwP5A_reference : Reference
_ACDO8A05Ed2ffdcmFWIZ3A_transition : Transition
_ACDO8A05Ed2ffdcmFWIZ3A-Transition-event_event
_aHrmIJOjEd2MX9xrFbwP5A_businessObject : Busines
_bfIWsJOiEd2MX9xrFbwP5A_businessObject : Busines
_bfpeoJOjEd2MX9xrFbwP5A_businessObject : Busines
_BuW_wA05Ed2ffdcmFWIZ3A_transition : Transition
_BuW_wA05Ed2ffdcmFWIZ3A-Transition-event_event
_DpeBkA05Ed2ffdcmFWIZ3A_transition : Transition
_DpeBkA05Ed2ffdcmFWIZ3A-Transition-event_event
_en6aoJOkEd2MX9xrFbwP5A_businessObject : Busine
_I44swA05Ed2ffdcmFWIZ3A_transition : Transition
_I44swA05Ed2ffdcmFWIZ3A-Transition-event_event :
_izVSMJOkEd2MX9xrFbwP5A_reference : Reference
_KlEh8JOlEd2MX9xrFbwP5A_reference : Reference
_Lna3LpOiEd2MX9xrFbwP5A_businessObject : Busines
_Mq-G0BbHEd2oRv4dMo1XiA_reference : Reference
_PguL8BbHEd2oRv4dMo1XiA_reference : Reference
_tdo4wJOkEd2MX9xrFbwP5A_businessObject : Busine
_tZS2MJOiEd2MX9xrFbwP5A_reference : Reference
_xieAIJOkEd2MX9xrFbwP5A_reference : Reference
_YjSwUJOlEd2MX9xrFbwP5A_reference : Reference
_YY5JEJOjEd2MX9xrFbwP5A_businessObject : Busines
ATbundles.1-ATtypes.0-ATreferences.0_reference : F
ATbundles.1-ATtypes.1-ATitems.0-ATreferences.1_re
ATdialogs.0_dialog : Dialog
ATdialogs.0-ATcontent_eventTriggerWidget : EventTr
ATdialogs.0-ATcontent-ATwidgets.0_eventTriggerWid
ATdialogs.0-ATcontent-ATwidgets.0-ATbody_eventTri
ATdialogs.0-ATcontent-ATwidgets.0-ATbody-ATwidge
ATdialogs.0-ATcontent-ATwidgets.0-ATbody-ATwidge
ATdialogs.0-ATcontent-ATwidgets.0-ATbody-ATwidge
```

```
State Machine
Action
Businessobjects
  id: de.feasiple.salesscenario.businessobject
  generators
    java
Context
Dialog
  id: de.feasiple.salesscenario.dialog
  generators
    jpdl
    scxml
View
  id: de.feasiple.salesscenario.view
  generators
    swt
```

Source Type	Relation	Target Type
Slot	slot_is_represented_by_ui...	Button.caption
Slot	slot_is_represented_by_ui...	Header.title
Slot	slot_is_represented_by_ui...	TableColumn.title
Slot	slot_is_represented_by_ui...	ButtonPanelItem.caption
Slot	slot_is_represented_by_ui...	Label.caption
Slot	slot_is_represented_by_ui...	TextField.value
StateMachine	is_lifecycle_of	BusinessObject
Group.title	uicaption_represents_slot	Slot
Button.caption	uicaption_represents_slot	Slot
Header.title	uicaption_represents_slot	Slot
TableColumn.title	uicaption_represents_slot	Slot
ButtonPanelItem.caption	uicaption_represents_slot	Slot
Label.caption	uicaption_represents_slot	Slot
TextField.value	uicaption_represents_slot	Slot

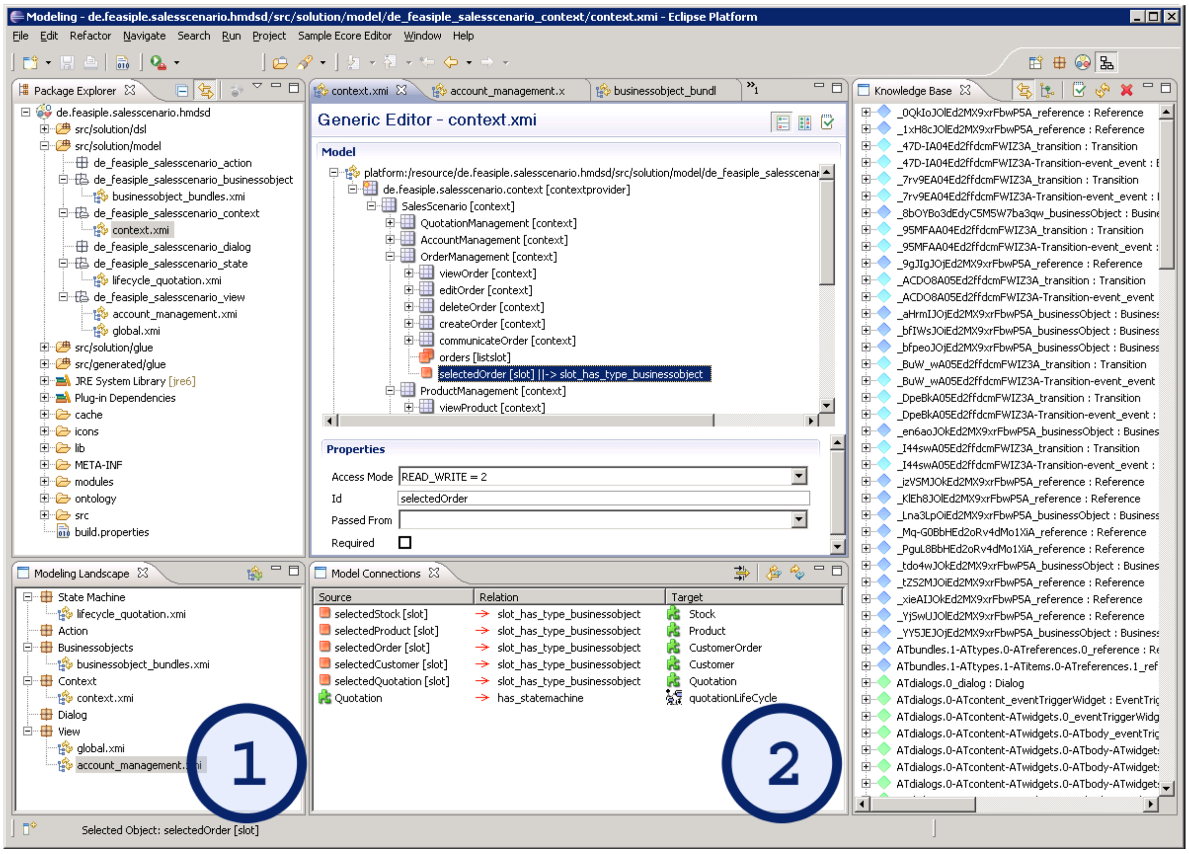
Figure 7.12: *The project engineering perspective*

Figure 7.13 the *modeling landscape view* is depicted. Here, the modeler gets information about all available modeling resources in his workbench. Area (2) highlights the *model connections view*. This view gives insight about actually instantiated inter-model connections. Hence, it complements the language composition view of the project engineering perspective that summarizes M2 level connections with information about connections on level M1. Like the views presented above, further explanations are given in the next section.

Moreover, to round up the indication of inter-model references we extended the generic editor to annotate tree nodes of depicted models with additional reference information. Thus, the selected node of the model shown in the perspective's center ("context.xmi") is extended with the string "|| -> slot_has_type_businessobject" to indicate that the corresponding model element is related with the inter-language connection slot_has_type_businessobject.

In summary, the modeling perspective does not add significant functionality to the default Eclipse modeling workbench and this is what we have actually intended. Besides the preliminary development tasks that are necessary to integrate employed DSLs on level M2, the actual modeling phase should be only transparently supported without imposing any additional practice efforts. Modelers that are familiar with a certain modeling technology shall be able to unfold their expertise without any obstacles. The additional views and editor adjustments are only subtle enhancements that come into play when inter-model references are to be maintained.

Finally, the modeling perspective is also the development environment where reasoner results, such as consistency checks or guidance proposals, are presented to the modeler. How these advanced feature are implemented in our is elucidated in the Section 7.3.



```
de.feasiple.salesscenario.hmdsd
  src/solution/dsl
  src/solution/model
    de_feasiple_salesscenario_action
    de_feasiple_salesscenario_businessobject
      businessobject_bundles.xmi
    de_feasiple_salesscenario_context
      context.xmi
    de_feasiple_salesscenario_dialog
    de_feasiple_salesscenario_state
      lifecycle_quotation.xmi
    de_feasiple_salesscenario_view
      account_management.xmi
      global.xmi
  src/solution/glue
  src/generated/glue
  JRE System Library [jre6]
  Plug-in Dependencies
  cache
  icons
  lib
  META-INF
  modules
  ontology
  src
  build.properties
```

```
platform:/resource/de.feasiple.salesscenario.hmdsd/src/solution/model/de_feasiple_salesscenar
  de.feasiple.salesscenario.context [contextprovider]
    SalesScenario [context]
      QuotationManagement [context]
      AccountManagement [context]
      OrderManagement [context]
        viewOrder [context]
        editOrder [context]
        deleteOrder [context]
        createOrder [context]
        communicateOrder [context]
        orders [listslot]
        selectedOrder [slot] ||-> slot_has_type_businessobject
      ProductManagement [context]
        viewProduct [context]
```

Access Mode	READ_WRITE = 2
Id	selectedOrder
Passed From	
Required	☐

```
_0QkIoJOlEd2MX9xrFbwP5A_reference : Reference
_1xH8cJOlEd2MX9xrFbwP5A_reference : Reference
_47D-IA04Ed2ffdcmFWIZ3A_transition : Transition
_47D-IA04Ed2ffdcmFWIZ3A-Transition-event_event : E
_7rv9EA04Ed2ffdcmFWIZ3A_transition : Transition
_7rv9EA04Ed2ffdcmFWIZ3A-Transition-event_event : I
_8bOYBo3dEdyC5M5W7ba3qw_businessObject : Busine
_95MFAA04Ed2ffdcmFWIZ3A_transition : Transition
_95MFAA04Ed2ffdcmFWIZ3A-Transition-event_event :
_9gJIgJOjEd2MX9xrFbwP5A_reference : Reference
_ACDO8A05Ed2ffdcmFWIZ3A_transition : Transition
_ACDO8A05Ed2ffdcmFWIZ3A-Transition-event_event
_aHrmIJOjEd2MX9xrFbwP5A_businessObject : Business
_bfIWsJOiEd2MX9xrFbwP5A_businessObject : Business
_bfpeoJOjEd2MX9xrFbwP5A_businessObject : Business
_BuW_wA05Ed2ffdcmFWIZ3A_transition : Transition
_BuW_wA05Ed2ffdcmFWIZ3A-Transition-event_event
_DpeBkA05Ed2ffdcmFWIZ3A_transition : Transition
_DpeBkA05Ed2ffdcmFWIZ3A-Transition-event_event :
_en6aoJOkEd2MX9xrFbwP5A_businessObject : Busines
_I44swA05Ed2ffdcmFWIZ3A_transition : Transition
_I44swA05Ed2ffdcmFWIZ3A-Transition-event_event :
_izVSMJOkEd2MX9xrFbwP5A_reference : Reference
_KlEh8JOlEd2MX9xrFbwP5A_reference : Reference
_Lna3LpOiEd2MX9xrFbwP5A_businessObject : Business
_Mq-G0BbHEd2oRv4dMo1XiA_reference : Reference
_PguL8BbHEd2oRv4dMo1XiA_reference : Reference
_tdo4wJOkEd2MX9xrFbwP5A_businessObject : Busines
_tZS2MJOiEd2MX9xrFbwP5A_reference : Reference
_xieAIJOkEd2MX9xrFbwP5A_reference : Reference
_YjSwUJOlEd2MX9xrFbwP5A_reference : Reference
_YY5JEJOjEd2MX9xrFbwP5A_businessObject : Business
ATbundles.1-ATtypes.0-ATreferences.0_reference : Re
ATbundles.1-ATtypes.1-ATitems.0-ATreferences.1_ref
ATdialogs.0_dialog : Dialog
ATdialogs.0-ATcontent_eventTriggerWidget : EventTrig
ATdialogs.0-ATcontent-ATwidgets.0_eventTriggerWidg
ATdialogs.0-ATcontent-ATwidgets.0-ATbody_eventTrig
ATdialogs.0-ATcontent-ATwidgets.0-ATbody-ATwidget:
ATdialogs.0-ATcontent-ATwidgets.0-ATbody-ATwidget:
ATdialogs.0-ATcontent-ATwidgets.0-ATbody-ATwidget:
```

```
State Machine
  lifecycle_quotation.xmi
Action
Businessobjects
  businessobject_bundles.xmi
Context
  context.xmi
Dialog
View
  global.xmi
  account_management.xmi
```

Source	Relation	Target
selectedStock [slot]	→ slot_has_type_businessobject	Stock
selectedProduct [slot]	→ slot_has_type_businessobject	Product
selectedOrder [slot]	→ slot_has_type_businessobject	CustomerOrder
selectedCustomer [slot]	→ slot_has_type_businessobject	Customer
selectedQuotation [slot]	→ slot_has_type_businessobject	Quotation
Quotation	→ has_statemachine	quotationLifeCycle

Selected Object: selectedOrder [slot]

Figure 7.13: *The modeling perspective*

7.1.3.4 Views

After the introduction of the three HybridMDSD perspectives that arrange several development views, these views are presented in more detail within this section.

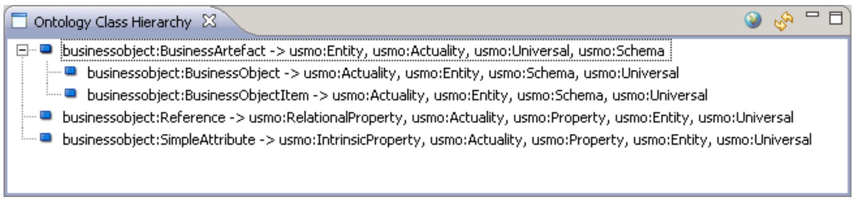
Ontology Class Hierarchy

The ontology class hierarchy view visualizes the taxonomy of ontologies. Figure 7.14 shows a screenshot. The view is updated according to the currently selected project in the package explorer. Thus, in case a language module project is selected the corresponding language ontology is shown and if a semantic connector project is selected the corresponding project ontology is shown. The view's menu bar contains two items besides the default items for view minimization and maximization. It is located in the upper right corner.

The first item (indicated by the globe icon) allows to switch between the two possible visualization modes: *local* and *global*. The local view shows the taxonomy of the actual ontology only without taking imported classes into account. Hence, the hierarchy tree shows only ontology classes that are directly declared. The super classes of these ontology classes are then represented as comma separated list behind the class name, as illustrated in Figure 7.14 (a).

In contrast, the global view also visualizes imported taxonomies. Hence, the super classes of directly declared types also appear in the hierarchy tree, as illustrated in Figure 7.14 (b). For better a separation of imported and directly declared classes, we employ two different icons for both nodes. Thus, imported nodes are indicated by pale node icons while declared nodes are more colorful.

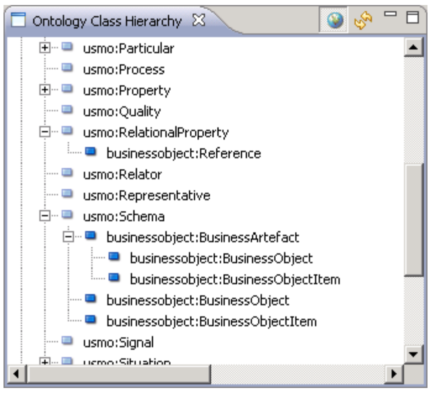
The reader might wonder why the ontology taxonomy depicted in Figure 7.14 contains du-



Ontology Class Hierarchy

```
businessobject:BusinessArtefact -> usmo:Entity, usmo:Actuality, usmo:Universal, usmo:Schema
    businessobject:BusinessObject -> usmo:Actuality, usmo:Entity, usmo:Schema, usmo:Universal
    businessobject:BusinessObjectItem -> usmo:Actuality, usmo:Entity, usmo:Schema, usmo:Universal
businessobject:Reference -> usmo:RelationalProperty, usmo:Actuality, usmo:Property, usmo:Entity, usmo:Universal
businessobject:SimpleAttribute -> usmo:IntrinsicProperty, usmo:Actuality, usmo:Property, usmo:Entity, usmo:Universal
```

(a) Ontology hierarchy view in local mode



Ontology Class Hierarchy

```
usmo:Particular
usmo:Process
usmo:Property
usmo:Quality
usmo:RelationalProperty
    businessobject:Reference
usmo:Relator
usmo:Representative
usmo:Schema
    businessobject:BusinessArtefact
        businessobject:BusinessObject
        businessobject:BusinessObjectItem
    businessobject:BusinessObject
    businessobject:BusinessObjectItem
usmo:Signal
usmo:Situation
```

(b) Ontology hierarchy view in global mode

Figure 7.14: *Ontology hierarchy view*

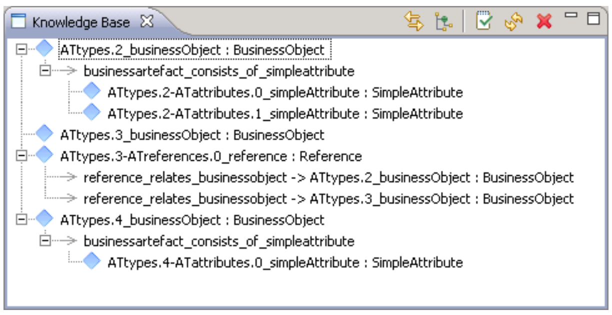
plicate nodes. For example, the ontology class `BusinessObject` appears as subclass of `BusinessArtefact` and `Schema` likewise. Now, this is actually caused by the reasoner settings of the HybridMDSD framework. The employed setup for the illustrated hierarchy view advices the reasoner to use a transitive inferencer that computes additional entailments, e.g., concerning inheritance relationships. Thus, although the shown language ontology only declares the `BusinessObject` class to be subclass of `BusinessArtefact`, the reasoner entails further superclasses of `BusinessObject`, such as `Schema`. Corresponding reasoner settings can be adjusted to the user's needs.

Knowledgebase

The knowlegdebase view shows the contents of the ontology knowledgebase that belongs to the currently selected project. Like the ontology hierarchy view also the knowledgebase view is adjusted when the project selection changes. The view's menu bar is divided into two sections, as depicted in Figure 7.15. The section of the right side contains (1) a *validation button* that causes the reasoner to apply consistency checks based on underlying DL restrictions and integrity rules, (2) a *refresh button* that causes the (re-)execution of the space bridge-based mapping process, and (3) a *deletion button* that causes the removal of all knowledgebase contents.

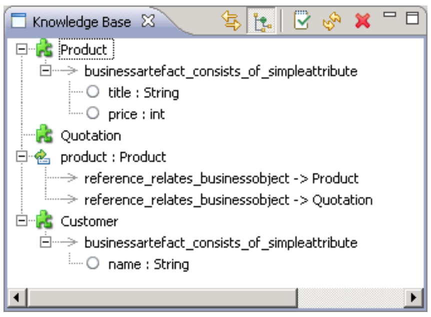
The menu bar's left section contains (1) the *connect mode button* and (2) the *editor rendering mode button*. The first button is explained below. What the second one affects is illustrated in Figure 7.15 (a) and (b), respectively. Figure 7.15 (a) shows all knowledgebase individuals with their plain identifiers followed by a colon and the name of the directly instantiated ontology class. Consequently, the rendered tree nodes are rather difficult to read and the entire tree has rather less value for developers.

This is why we introduced the *editor rendering mode button* that switches the tree node visualization. The result is illustrated in Figure 7.15 (b). The toggle button advices the HybridMDSD framework to extract and apply the domain-specific rendering from the employed model editor. To facilitate a comparison of both visualizations we attached the source model for the depicted



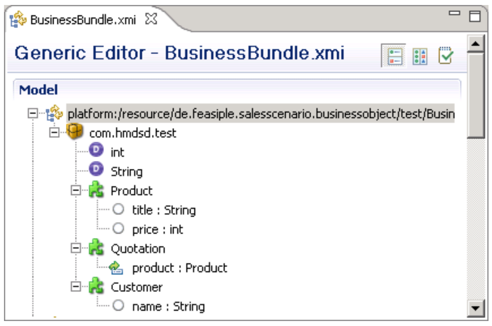
```
Knowledge Base
ATtypes.2_businessObject : BusinessObject
  businessartefact_consists_of_simpleattribute
    ATtypes.2-ATattributes.0_simpleAttribute : SimpleAttribute
    ATtypes.2-ATattributes.1_simpleAttribute : SimpleAttribute
ATtypes.3_businessObject : BusinessObject
ATtypes.3-ATreferences.0_reference : Reference
  reference_relates_businessobject -> ATtypes.2_businessObject : BusinessObject
  reference_relates_businessobject -> ATtypes.3_businessObject : BusinessObject
ATtypes.4_businessObject : BusinessObject
  businessartefact_consists_of_simpleattribute
    ATtypes.4-ATattributes.0_simpleAttribute : SimpleAttribute
```

(a) View with plain node names



```
Knowledge Base
Product
  businessartefact_consists_of_simpleattribute
    title : String
    price : int
Quotation
product : Product
  reference_relates_businessobject -> Product
  reference_relates_businessobject -> Quotation
Customer
  businessartefact_consists_of_simpleattribute
    name : String
```

(b) View with editor-specific node names



```
BusinessBundle.xmi
Generic Editor - BusinessBundle.xmi
Model
platform:/resource/de.feasiple.salesscenario.businessobject/test/Busin
  com.hmdsd.test
    int
    String
    Product
      title : String
      price : int
    Quotation
      product : Product
    Customer
      name : String
```

(c) Source model for the knowledgebase depicted in (a) and (b)

Figure 7.15: *Knowledgebase view*

knowledgebase in Figure 7.15 (c). Altogether, the three illustrations show very nice (1) which model elements are actually mapped into the ontology TS and (2) the differences between the structures of both TSs.

The *editor rendering mode button* is a toggle that causes the HybridMDSD framework to provide an appropriate connection context menu entry in case a model element in a model editor is activated with a right mouse click. Thus, the context menu contains a menu item "Connection" that unfolds a submenu which contains all availabe inter-language relations that are appropriate for the selected element. The submenu, in turn, contains further menu items that list all possible reference counterparts that can be found in the modeling landscape, as illustrated in Figure 7.16.

The entire resolution processes is exclusively performed on the basis of the semantic infrastructure. Hence, for a selected model element the mapped knowledgebase individual is queried. This individual is used to search for matching inter-DSL relations. Then, the knowledgebase is queried for individuals that suffice the corresponding range types of returned relations. Finally, these individuals are used to derive mapped model elements, which are rendered in the context menu.

Knowledgebase Graph

For better illustration of the knowledgebase structures we created a graph view for OWL-based files on the basis of the *prefuse Information Visualization Toolkit* [oD09]. In correspondence to the common knowledgebase view, the graph view renders each knowldge base individual as diamond node. The relations between individuals are rendered as edges. Both, nodes and edges are equipped with labels that indicate the individual and relation name, respectively. Figure 7.17 shows the result.

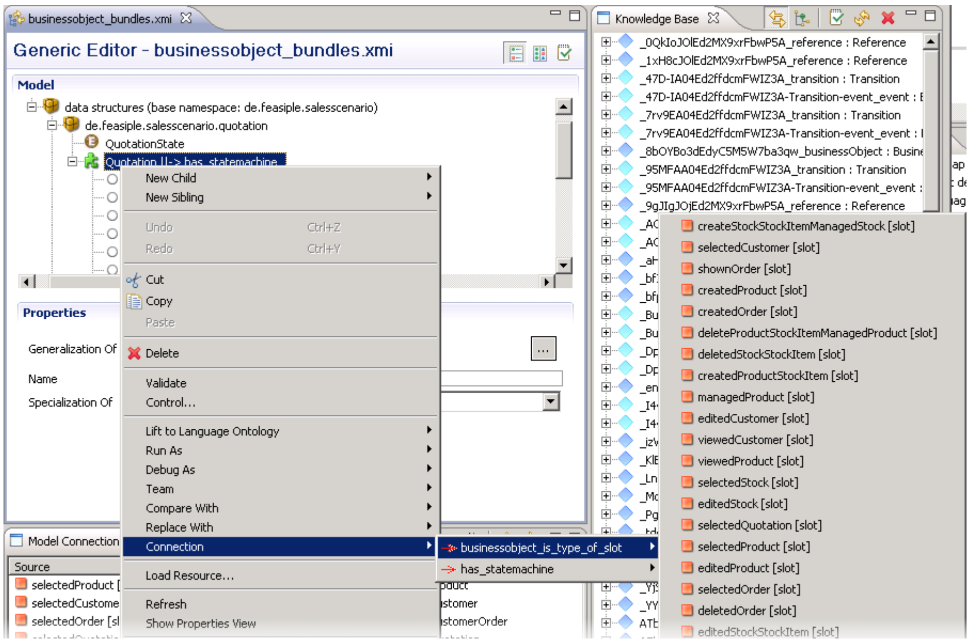

businessobject_bundles.xmi
Generic Editor - businessobject_bundles.xmi
Model
data structures (base namespace: de.feasiple.salesscenario)
de.feasiple.salesscenario.quotation
QuotationState
Quotation |-> has_statemachine
New Child
New Sibling
Undo Ctrl+Z
Redo Ctrl+Y
Cut
Copy
Paste
Delete
Validate
Control...
Lift to Language Ontology
Run As
Debug As
Team
Compare With
Replace With
Connection
Load Resource...
Refresh
Show Properties View
businessobject_is_type_of_slot
has_statemachine
Properties
Generalization Of
Name
Specialization Of
Model Connection
Source
selectedProduct [
selectedCustome
selectedOrder [sl
Knowledge Base
_0QkIoJOlEd2MX9xrFbwP5A_reference : Reference
_1xH8cJOlEd2MX9xrFbwP5A_reference : Reference
_47D-IA04Ed2ffdcmFWIZ3A_transition : Transition
_47D-IA04Ed2ffdcmFWIZ3A-Transition-event_event : E
_7rv9EA04Ed2ffdcmFWIZ3A_transition : Transition
_7rv9EA04Ed2ffdcmFWIZ3A-Transition-event_event : I
_8bOYBo3dEdyC5M5W7ba3qw_businessObject : Busine
_95MFAA04Ed2ffdcmFWIZ3A_transition : Transition
_95MFAA04Ed2ffdcmFWIZ3A-Transition-event_event :
_9gJIgJOjEd2MX9xrFbwP5A_reference : Reference
createStockStockItemManagedStock [slot]
selectedCustomer [slot]
shownOrder [slot]
createdProduct [slot]
createdOrder [slot]
deleteProductStockItemManagedProduct [slot]
deletedStockStockItem [slot]
createdProductStockItem [slot]
managedProduct [slot]
editedCustomer [slot]
viewedCustomer [slot]
viewedProduct [slot]
selectedStock [slot]
editedStock [slot]
selectedQuotation [slot]
selectedProduct [slot]
editedProduct [slot]
selectedOrder [slot]
deletedOrder [slot]
editedStockStockItem [slot]

Figure 7.16: *Creating inter-model references with the connection context menu*

Although the prefuse framework provides a huge set of features to control the layout and appearance of graph structures, we stoped the elaboration of our view in the very beginning. This was because we witnessed quite early that a graph-based visual representation of already very small project knowledgebases results in confusingly huge structures that are difficult to interpret. Hence, for suitable tool support it would have been necessary to develop further visualization concepts that go beyond a simple one-to-one projection of individuals and relations. But this was clearly out of scope of this thesis.

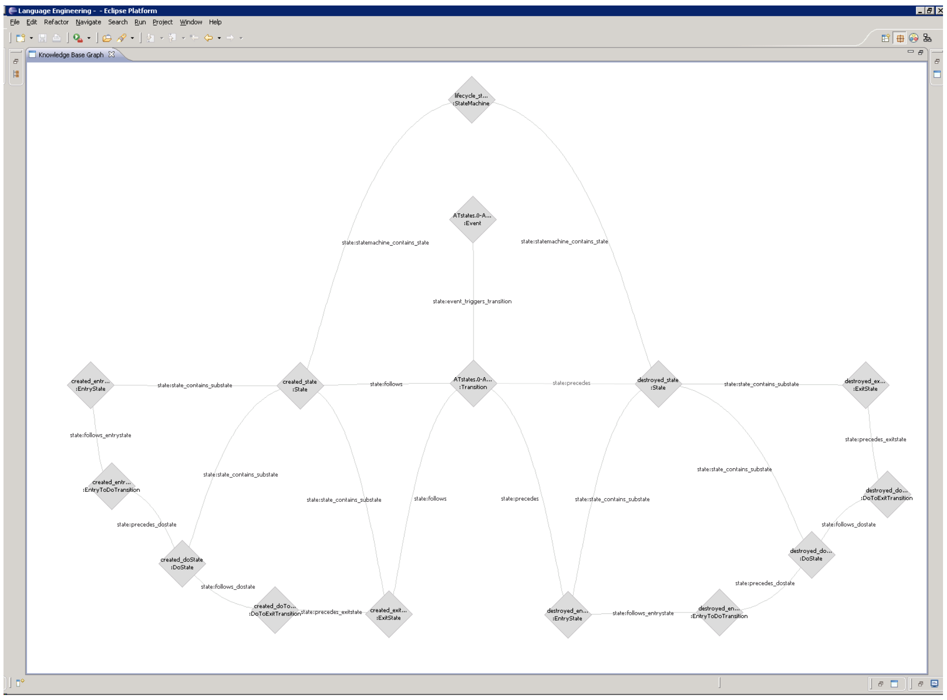
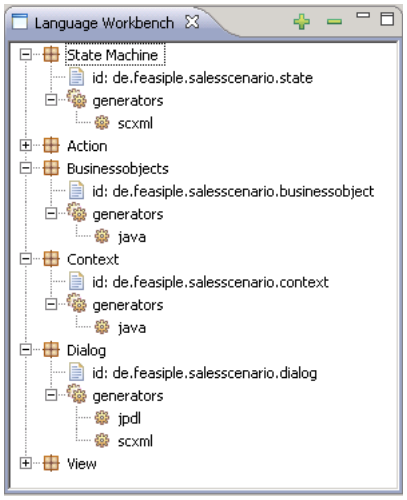

Language Engineering - - Eclipse Platform
File Edit Refactor Navigate Search Run Project Window Help
Knowledge Base Graph
lifecycle_st... :StateMachine
ATstates.0-A... :Event
state:statemachine_contains_state
state:event_triggers_transition
created_entr... :EntryState
created_state :State
ATstates.0-A... :Transition
destroyed_state :State
destroyed_ex... :ExitState
state:state_contains_substate
state:follows
state:precedes
state:follows_entrystate
state:precedes_exitstate
created_entr... :EntryToDoTransition
destroyed_do... :DoToExitTransition
state:precedes_dostate
state:follows_dostate
created_doState :DoState
destroyed_do... :DoState
created_doTo... :DoToExitTransition
created_exit... :ExitState
destroyed_en... :EntryState
destroyed_en... :EntryToDoTransition

Figure 7.17: *Knowledgebase graph view*

Language Workbench

The language workbench view belongs to the project engineering perspective and allows to get an overview of the language modules that are composed within a semantic connector project. To this end the view is — like the views before as well — project selection sensitive. Hence, in case no semantic connector project is selected it appears grayed out and is disabled. With the language workbench view the developer is enabled to import or remove language modules from a connector project. Here, a dialog similar to the wizard page of the project creation wizard comes into play (cf. Figure 7.8). A newly imported language module is processed by (1) importing the module's language ontology into the connector's project ontology, (2) creating appropriate folders below the project resource, and (3) copying or referencing corresponding metamodeling resources.



```
Language Workbench
State Machine
    id: de.feasiple.salesscenario.state
    generators
        scxml
Action
Businessobjects
    id: de.feasiple.salesscenario.businessobject
    generators
        java
Context
    id: de.feasiple.salesscenario.context
    generators
        java
Dialog
    id: de.feasiple.salesscenario.dialog
    generators
        jpdl
        scxml
View
```

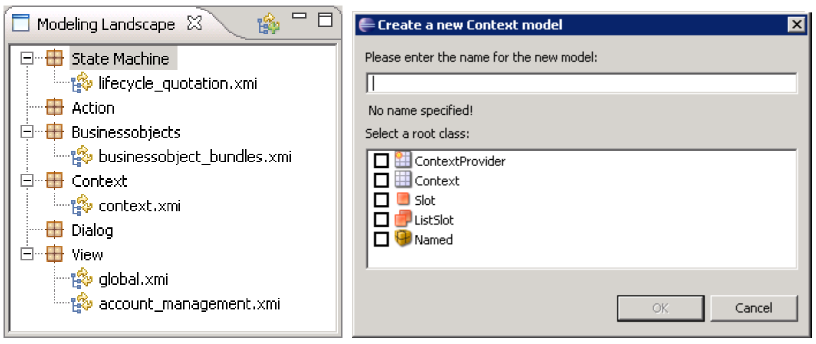
Figure 7.18: *Language workbench view*

Besides the import and remove functionality the language workbench view illustrates several details about available modules. As shown in Figure 7.18, for each language module a particular tree node is rendered, which contains child nodes that indicate the module's id and provided code generators.

Modeling Landscape

The modeling landscape view is part of the modeling perspective and shall provide control over available modeling resources (see Figure 7.19 (a)). To this end, the view shows a tree widget whos root nodes represent imported language modules, as in the language workbench view above. Below each module's node all existing modeling resources are listed. Here, the framework simply gets all files that are contained in the folder that is watched by the space bridge of each language module. The view supports mouse double-clicks that open selected resources in the default model editor.

The one and only item in the view's menu bar allows to create new models for selected language modules. Therefore, an appropriate dialog pops up that lists all metamodel classes that can be instantiated (see Figure 7.19 (b)). Additonally, the dialog prompts for a name for the model element to create. Once the dialog is confirmed, a new model resource is created with an instance of the selected metamodel class as root element.


Modeling Landscape

- State Machine
  - lifecycle_quotation.xmi
- Action
- Businessobjects
  - businessobject_bundles.xmi
- Context
  - context.xmi
- Dialog
- View
  - global.xmi
  - account_management.xmi

Create a new Context model

Please enter the name for the new model:

No name specified!

Select a root class:

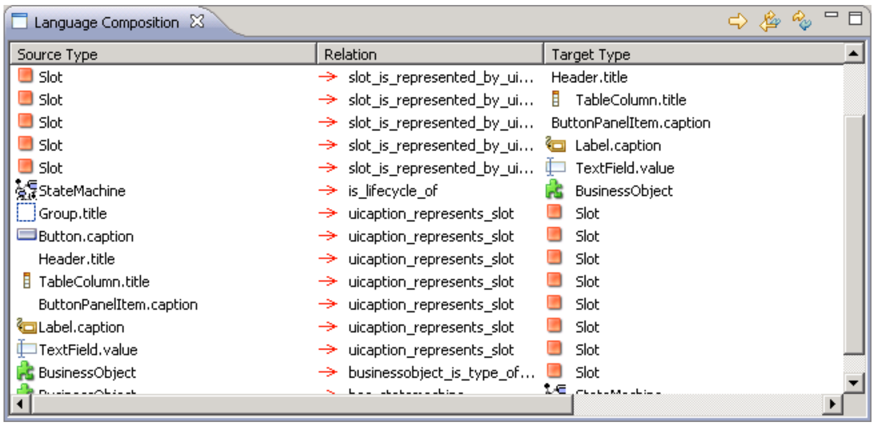
- ContextProvider
- Context
- Slot
- ListSlot
- Named

(a) The plain view (b) Create new model dialog

Figure 7.19: *Modeling landscape view*

Language Composition

The language composition view supports the project engineering phase and is arranged in the corresponding perspective. As shown in Figure 7.20, the view lists all available inter-language connections by presenting all properties that are defined in the project ontology. Here, in case an inter-DSL relation is specified with more than one range type, one line per range type is produced. Like in the model connection context menu (see above), not the bare ontology classes are illustrated but rather the metamodel types that map to these classes.


Language Composition

Source Type	Relation	Target Type
Slot	slot_is_represented_by_ui...	Header.title
Slot	slot_is_represented_by_ui...	TableColumn.title
Slot	slot_is_represented_by_ui...	ButtonPanelItem.caption
Slot	slot_is_represented_by_ui...	Label.caption
Slot	slot_is_represented_by_ui...	TextField.value
StateMachine	is_lifecycle_of	BusinessObject
Group.title	uicaption_represents_slot	Slot
Button.caption	uicaption_represents_slot	Slot
Header.title	uicaption_represents_slot	Slot
TableColumn.title	uicaption_represents_slot	Slot
ButtonPanelItem.caption	uicaption_represents_slot	Slot
Label.caption	uicaption_represents_slot	Slot
TextField.value	uicaption_represents_slot	Slot
BusinessObject	businessobject_is_type_of...	Slot

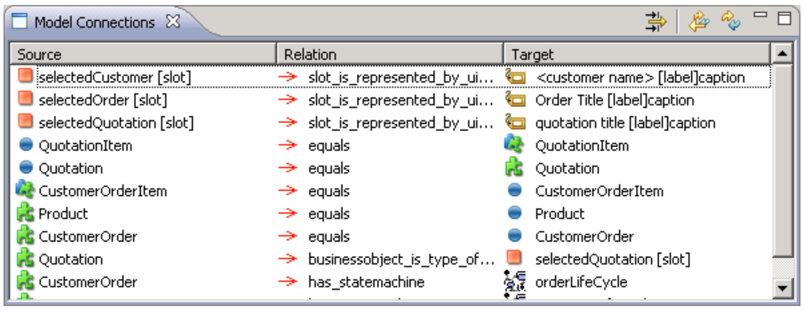
Figure 7.20: *Language composition view*

The first item in the view's menu bar facilitates the creation of a new composition generator for the selected inter-DSL connection. Clicking on this button entails the creation of a new composition template in the corresponding source folder "src\solution\glue", as shown in Listing 6.2 of Chapter 6. Hence, the composition expert gains control about inter-language connections on abstract and concrete level.

The second and third menu item are used to invoke specified composition generators. To this end, the former button invokes the composer that is associated with the current selection, while the latter button invokes all composition generators that are listed. Invoking a composition generator produces actual glue code for all instantiated inter-language connections in the modeling landscape.

Model Connections

The model connections view is arranged in the modeling perspective. This view facilitates the maintenance of inter-model references in the modeling landscape. Therefore, it lists all connections that were created between the models of different language modules. Figure 7.21 shows a screenshot of the view. The view's menu bar consists of three buttons. The leftmost button switches the view into the selection-sensitive filter mode. In this mode, only those inter-model connections are shown that involve currently selected model elements in the model editor. Hence, only those references are listed where either source or target modeling contstruct are contained.


Model Connections

Source	Relation	Target
selectedCustomer [slot]	slot_is_represented_by_ui...	<customer name> [label]caption
selectedOrder [slot]	slot_is_represented_by_ui...	Order Title [label]caption
selectedQuotation [slot]	slot_is_represented_by_ui...	quotation title [label]caption
QuotationItem	equals	QuotationItem
Quotation	equals	Quotation
CustomerOrderItem	equals	CustomerOrderItem
Product	equals	Product
CustomerOrder	equals	CustomerOrder
Quotation	businessobject_is_type_of...	selectedQuotation [slot]
CustomerOrder	has_statemachine	orderLifeCycle

Figure 7.21: *Model connections view*

Menu bar button two and three are already known from the previous view. They allow to control the instantiation of composition generators. In contrast to the language composition view, this view acts on actual modeling level rather than language level. Hence, while the language composition view allowed to invoke composition generators only for dedicated inter-model connection *types*, i.e., inter-language connections, the model connections view enables composition generator invocation for each and every reference *instance*.

7.2 Case Study: The SalesScenario

In the last section we comprehensively presented the prototypical implementation of our HybridMDSD approach to multi-domain engineering. Now, within this section we demonstrate the application of this prototype on the basis of a sophisticated industrial-scale case study. We have developed the case study in context of the german and european research projects feasiPLe [Fea07] and AMPLE [Con07a]. Both projects aim at an integration of model-driven and product line technologies in software engineering.

We aligned the structure of this section to the HybridMDSD development phases, presented in Section 6.3. To this end, the iterative nature of this process hampers a entirely sequential presentation within a text document. Therefore, it was necessary to split the presentation of the project engineering phase into (1) requirement analysis and scoping (see Section 7.2.1) and (2) language integration (see Section 7.2.3).

Please note that the presented case study was already previously introduced within several research publications [LG08b, PCG^{+}08]. These publications were co-authored by the author of this thesis. Therefore, the following sections might share content.

7.2.1 Project Engineering: Requirements

Recall from Section 6.3.4, that HybridMDSD project engineering starts with the selection of language modules. Here, domain and platform requirements are identified and elaborated. Once the requirements are clarified, platform decision must be rendered which sets particular requirements to the generator imlementations of necessary lanuage modules.

7.2.1.1 Language Module Selection

To summarize the requirement analysis activities, we start with an introduction to the domain of our case study. Succeedingly, we shed light on our considerations concerning platform-centric requirements.

Domain Requirements

Our case study demonstrates software engineering in the domain of enterprise software. In this respect, the corresponding software product line is an implementation of the briefly explained motivational example introduced in Section 3.1. Hence, our case study demonstrates business application engineering around *Enterprise Resource Planning (ERP)*. This domain is an ideal condidate to demonstrate variability since corresponding solutions must be adapted and customized to a particular company (no two companies are the same). To reduce the complexity for the sake of conciseness we focused on a particular sub-domain — *Customer Relationship Management (CRM)* — combined with some parts of PLM, SCM, and SRM. We coined the case study project *SalesScenario* to clearly distinguish from CRM as explained by Buck-Emden and Zencke in [BEZ04].

In order to give a concise introduction to common application scenarios in the domain of interest, we describe the overall sales process and its individual components that are to be implemented below. To this end, emphasized words denote candidates for modularization which led to the case study's feature model, presented below (see Figure 7.50).

Step 1: To start business with a potential customer, the corresponding master data, such as customer name and address, budget estimation and a description of the sales opportunity and its time frame is saved to a customer profile (*Prospect*) by using the functionalities of *Account Management*.

Step 2: After subsequent evaluation and go/no go decision handling by sales management, another employee of the sales office creates a quotation (i.e., offer) using the *Quotation* functionalities and configures a quotation template based on the sales opportunity data. A potential discount is included into such a quotation by using *Pricing Strategy*. The calculation strategy is based on the categorization of *Prospects* in a *Customer Group*, estimated sales volume, and sales probability, resulting in an overall *Customer Rating*.

Step 3: After the sales office has contacted the customer and received an order, the system automatically converts the quotation into an order upon mouse click using the corresponding *Sales Processing* functionality.

Step 4: To check the creditworthiness of the customer, an (optional) *Credit Check* is performed before creating a quotation as well as before accepting a sales order, by interacting with the *Payment* module.

Step 5: An *Availability Check* is performed to check for sufficient *Stock* and necessary capacities in the warehouse. In case of *Multiple Stocks*, meaning, delivery processes and storage of different goods depend on several warehouses, only those warehouses sufficiently close to the shipping address are included.

Step 6: If *Payment* is to be integrated into the sales process, it would be activated automatically upon creation of a binding sales order. Depending on the method of payment offered by the system and selected by the customer, at least one of the following feature options are realized: An automatic debit transfer from the customer's account can be triggered (*Payment Card*) or an invoicing document can be attached to the delivery to account for *Cash On Delivery*, or if payment is settled at a later stage, *Invoicing*.

Step 7: The order status is set to “sent” by an employee as soon as the order is delivered to the customer. Once an open invoice is paid by the customer, the order is marked as “paid” and “closed”. In case an open or already “sent” sales order is returned by the customer, carried out by *Returns*, an *Approval Process* is triggered, which involves sales management processes for commitment.

The description of all *Communication* channels has been omitted for the sake of clarity. Such instances are naturally related to mediation of quotations, invoices etc., whether sent by *e-Mail*, *Fax*, or *Letter*.

Platform Requirements

A detailed analysis of the process description, further domain information from [BEZ04], and the preliminarily identified feature candidates led to an initial architecture for the product line’s solution space (see Figure 7.22). The functional nature of above emphasized parts allowed the derivation of modular architecture blocks for main components. The interconnections between these blocks arose from the domain description and therein contained information about process-centric interdependencies. Additional parts, such as the user interface, came from technical considerations and corresponding stakeholders. The sketched architecture was a key milestone to estimate necessary solution space DSLs.

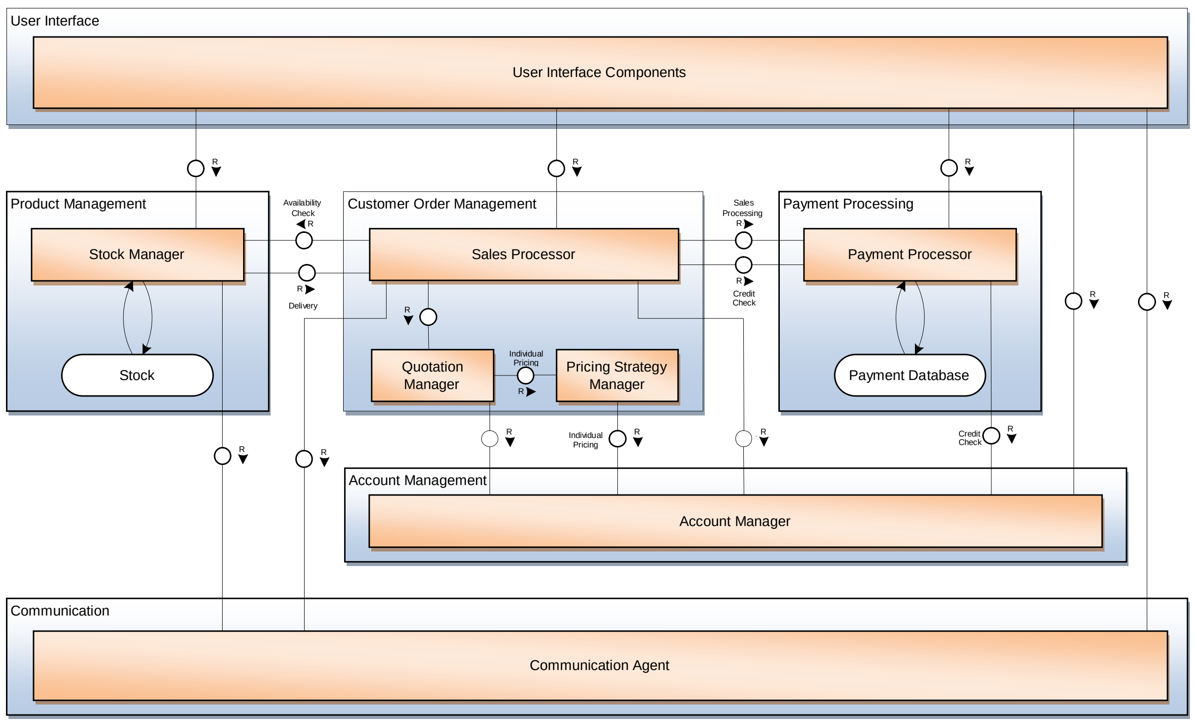


Figure 7.22: *SalesScenario architecture overview (from [LG08b])*

The depicted high-level architecture follows the visual guidelines of the FMC notation [TAW03]. It outlines the key entities and their interactions according to the functionality described above. Prominent are well modularizable, coherent features on the one hand and wide spread, crosscutting features on the other. Exemplary for coherent parts are *Stock Management* and *Payment Processing* components. Spreading components encapsulate non-obvious cross-cutting concerns by widely distributing implementational or invocational parts of their functionality. A proper visual distinction between the two feature types and their corresponding architecture modules respectively is not given in Figure 7.22. Nonetheless, one indication may be the number of communication connections of one block to others, where many connections prove cross-cutting functionality. A detailed introduction of the corresponding feature model and its features respectively, is given in Section 7.2.4.1.

Necessary Language Modules

In conclusion, the above stated requirement context led to the need of language modules for the following solution space areas.

Data Modeling: A business application depends heavily on data. This involves the storage of customer, product, order, and stock data.

State Modeling: The sales process description indicates that certain business entities may be in particular states depending on the process progress.

Action Modeling: Certain functionality, such as sending out mail or triggering availability checks, must be callable from different components.

User Interface Modeling: A user interface to interact with the target application is vital for the usage of final software products in general.

Context Modeling: To bridge the gap between holistic persistent storage and query-based data subsets in the UI, a session-based in-memory data model is required.

Pageflow Modeling: The sketched sales process illustrates how different tasks are sequentially chained. To represent such chains and dialog navigation in general we need an appropriate pageflow modeling language.

7.2.1.2 Platform Selection

Our software product line is a business application for product selling companies. Commonly, software applications in this domain are widely distributed, allowing multiple users and user groups to retrieve and modify different kinds of information. The distributed nature of business applications is a major technical constraint when searching for appropriate implementation means. In general, we had to make a selection between a *Rich-Client Platform (RCP)* with additional communication support and central database, and a web application platform. For two major reasons we decided to develop an RCP-based implementation, although this implied an increase of the final engineering effort, considering the amount of available web application frameworks that already provide convenient solutions.

First and foremost, the usage of an existing web application framework, such as Spring [Joh07] or JBoss Seam [JBS08], would conceal a significant amount of otherwise necessary implementation. Recall from Section 2.2.3.1, that web application frameworks are interpretive systems that provide and integrate several DSLs in internal implementations. Thus, even though we would implement our own DSLs and generate appropriate assets amenable by an underlying web application framework, an actual integration for produced assets would be superfluous. Instead, the integration is realized by and hidden in the employed framework. Now, this would be inappropriate for a validation of our approach because valuable insights concerning the challenges during language integration would be lost.

Second, considering the efforts during language engineering, web application frameworks massively hamper tryouts, testing, and initial validation. Testing a web application involves the deployment of comprehensive packages to corresponding servers. Only after deployment generated code can be verified. Because of necessary deployment steps the verification of dynamically developed generators becomes bulky and tedious, which unnecessarily impedes engineering in our context.

Concludingly, we selected a J2SE-based platform for the SalesScenario case study. To this end, we investigated which frameworks, libraries, or platforms were appropriate to implement the functionality encapsulated by each language module. Details about the resulting technology platform are given with the explanation of the modules in the next section.

7.2.2 Language Engineering

In the previous section we identified language modules that are necessary to implement the SalesScenario product line. In this section, we document on the language engineering phase that yielded these modules. Every language module is presented in a dedicated section. All of these sections follow the same structure and contain (1) a statistical overview, (2) an illustration of the module's metamodel, and (3) an illustration of its language ontology. The textual description briefly introduces corresponding assets and focuses on particularities we exposed during development.

7.2.2.1 Data Modeling

The language module that we introduce at first is a module that facilitates the modeling of data structures which are to be instantiated at runtime. Table 7.2 provides an appropriate overview.

Header	
Name	Businessobjects
ID	de.feasiple.salesscenario.businessobject
Metamodel & Generator	
Metamodel Classes	12
Generator Target Platform	Java POJOs
Domain-specific Framework (classes / LOC)	–
Generator Templates (count / LOC)	1 / 173
Semantic Infrastructure	
Element Handlers	4
Attribute/Reference Handlers	–
Language Ontology Classes	5
Language Ontology Properties (incl. rev.)	3 (6)

Table 7.2: *Overview data modeling language module*

In Figure 7.23, the ecore-based metamodel of the language module's DSL is shown. As depicted, it divides data modeling into simple DataTypes and more complex BusinessObjects, which are both contained in a global BusinessBundle resource. Business objects can further be composed of SimpleAttributes and References, where the former denotes elementary data type values and the latter declares referential relationships to other business objects. Concerning the visual syntax for this metamodel, we provided enhancements for the oAW generic editor for EMF models (cf. Section 7.1.3.1 and 7.1.3.3).

We equipped the data modeling DSL with a code generator that produces ordinary Java classes for each business object, with appropriate attributes of corresponding types. Figure 7.24 shows an exemplary BusinessObject in the generic editor mentioned above. For the depicted model the

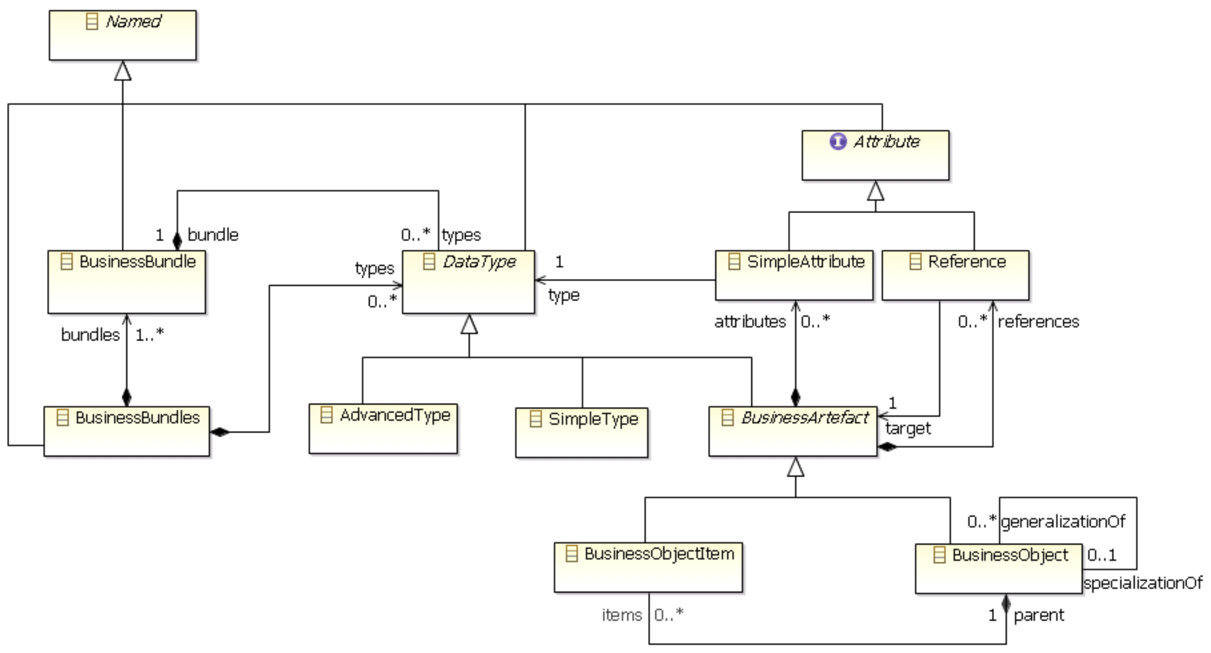


Figure 7.23: *Metamodel of the data modeling DSL*

generator produces, among others, the Java class shown in Listing 7.1. For brevity reasons, not all generated assets are listed. Additional Java classes, for instance, would be generated for the BusinessObjects Product and Customer, but also for the BusinessObjectItem QuotationItem.

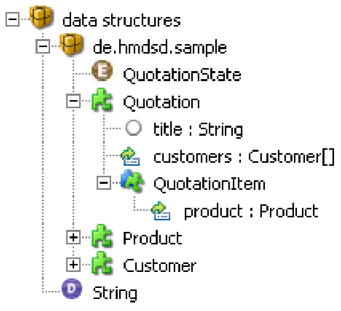

```
data structures
  de.hmdsd.sample
    QuotationState
    Quotation
      title : String
      customers : Customer[]
      QuotationItem
        product : Product
    Product
    Customer
  String
```

Figure 7.24: *An exemplary data model in graphical syntax*

An appropriate language ontology for the data modeling language is depicted in Figure 7.25. Recall from Section 6.3.3.3, that language ontologies capture the ontological commitment of a DSLs metamodel. For the sake of clarity we named corresponding ontology classes like the metamodel classes they represent.

Considering the language ontology for the data modeling DSL already reveals significant value that arises with the semantic foundation of the upper ontology USMO. Firstly, we obtain a precise semantics for containment relations, compared to the relatively poor definition provided by EMF and MOF, respectively. According to [BET08], the EMF containment relation "defines an ownership relation between objects". This semantics can be represented with contains property in USMO. However, the semantics of the EMF containment relation is not precise or unambiguous. Admittedly, EMF containment represents transitive non-sharable part-whole relationship but the relation between BusinessObjects and SimpleAttributes is even stronger. It rather denotes a composite aggregation in the sense of the UML. To this end, the USMO property consists_of provides appropriate semantics and relates the corresponding ontology classes accordingly.

```java
package de.hmdsd.sample;
// ...
public class Quotation {

	private Customer[] customers;

	private QuotationItem[] items;

	private String title;

	// ...
	public String getTitle() {
		return this.title;
	}

	public void setTitle(String title) {
		this.title = title;
	}
	// ...
}
```

Listing 7.1: *An excerpt of generated code for a data model*

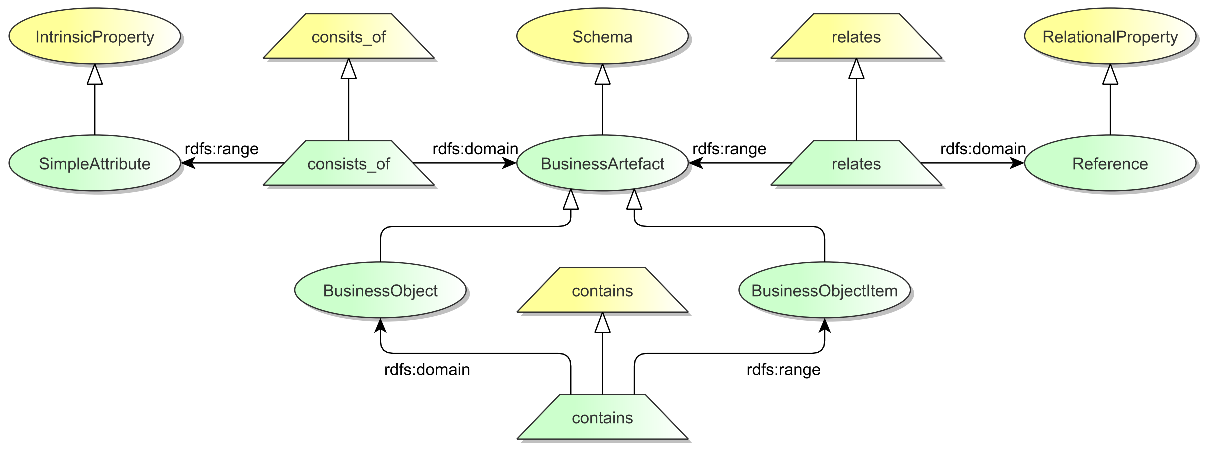


Figure 7.25: *Data modeling language ontology*

Another improvement of the explicitly modeled hidden semantics is the clear capture of the ontological type of the elements in the businessobject metamodel. This does not arise from the semantics of EMF. The subclass relation of language ontology classes to USMO universals prove that the data modeling DSL forms a *type* rather than a *token* language.

Finally, one interesting characteristic within the semantic mapping arises with respect to the representation of entity relations. Here, entity relations or references are represented as independent entities neither composed nor contained in related entities. Recall from Section 6.1, that this particularity is based on well-discussed semantics and helps to capture the genuine meaning of metamodeling constructs.

7.2.2.2 State Modeling

The second language module enables developers to define object behavior with simplified UML-like state diagrams. A summarizing overview is given in Table 7.3 that already reveals an interesting characteristic. Thus, the language ontology counts twice as many classes as the represented metamodel. The reason for this is explained below.

Header	
Name	State Machine
ID	de.feasiple.salesscenario.state
Metamodel & Generator	
Metamodel Classes	5
Generator Target Platform	Java + SCXML
Domain-specific Framework (classes / LOC)	–
Generator Templates (count / LOC)	1 / 121
Semantic Infrastructure	
Element Handlers	4
Attribute/Reference Handlers	1
Language Ontology Classes	11
Language Ontology Properties (incl. rev.)	13 (16)

Table 7.3: *Overview state modeling language module*

As shown in the module's metamodel in Figure 7.26, the DSL bundles different states in a global StateMachnine, which has a dedicated start state. Each State contains a number of Transitions to other states as well as optional Actions that are executed when a state is reached. Here, different Actions are executed when a State is entered, active, or left. Each transition is triggered by a particular event, which is provided as attribute of the Transition class and omitted in Figure 7.26, for brevity reasons.

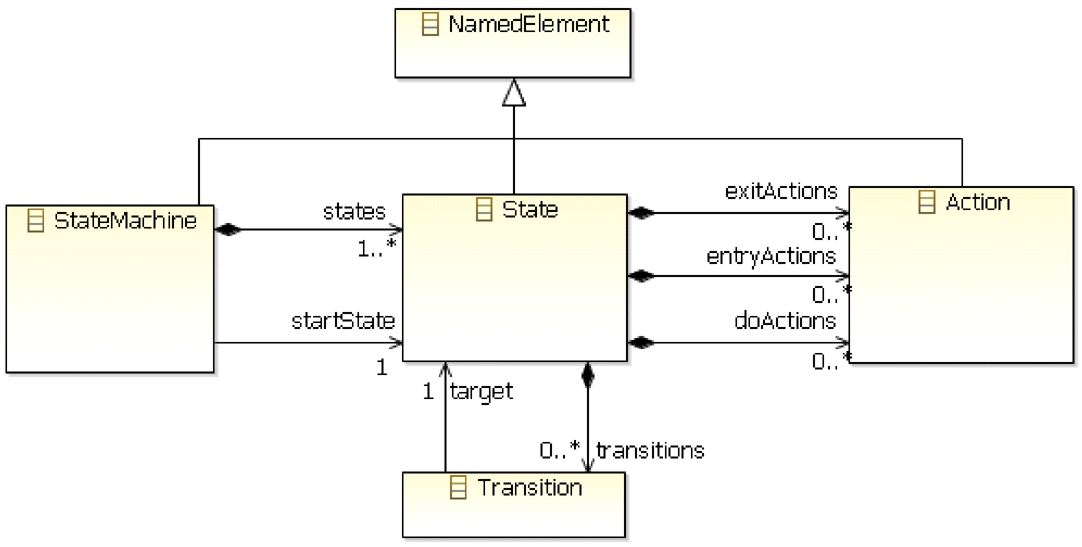


Figure 7.26: *Metamodel of the state modeling DSL*

The state modeling code generator produces one Java class per StateMachine instance, which is usable as facade and allows to trigger events that cause state transitions. To this end, the class inherits from a base class provided by the *Commons SCXML* framework [Fou09c]. The framework class is instantiated with an SCXML file that defines the actual state machines semantic and is generated as well. Listing 7.2 shows a simple state machine in textual syntax. Excerpts from the assets produced by the state machine code generator are shown in Listing 7.3.

```
statemachine quotationLifeCycle {
        namespace de.hmdsd.sample.behaviour .
        start created .

        // ...
        state created {
                on propose -> proposed .
        }

        state proposed {
                action sendMail .
        }
        // ...
}
```

Listing 7.2: *An excerpt of an exemplary state model in textual syntax*

```
package de.feasiple.salesscenario.quotation.life;

import org.apache.commons.scxml.env.AbstractStateMachine;

public class QuotationLifeCycle extends AbstractStateMachine {

        public static enum Event {
                DETAILSFILLED(
                        "quotationLifeCycle.detailsFilled"), PREPARE(
                        "quotationLifeCycle.prepare"), PROPOSE(
                        "quotationLifeCycle.propose"), ACCEPT(
                        "quotationLifeCycle.accept"), REJECT(
                        "quotationLifeCycle.reject"), ARCHIVE(
                        "quotationLifeCycle.archive"), ;
                // ...
        }

        public QuotationLifeCycle() {
                super(QuotationLifeCycle.class
                        .getResource("quotationlifecycle.xml"));
        }

        public void fireEvent(Event event) {
                this.fireEvent(event.getId());
        }

        public void proposed() {
                System.out.println("proposed state reached");
        }
        // ...
}
```

Listing 7.3: *An excerpt of generated code for a state model*

Figure 7.27 shows the language ontology for the state modeling DSL. As already shown for the data language, appropriate ontology classes are defined that, on the one hand, map to equally named metamodel classes and, on the other hand, specialize particular ontology classes from USMO. Interestingly, as stated above the language ontology contains far more ontology classes than the represented metamodel. This characteristic is justified with the straightforward and minimalistic metamodel contents.

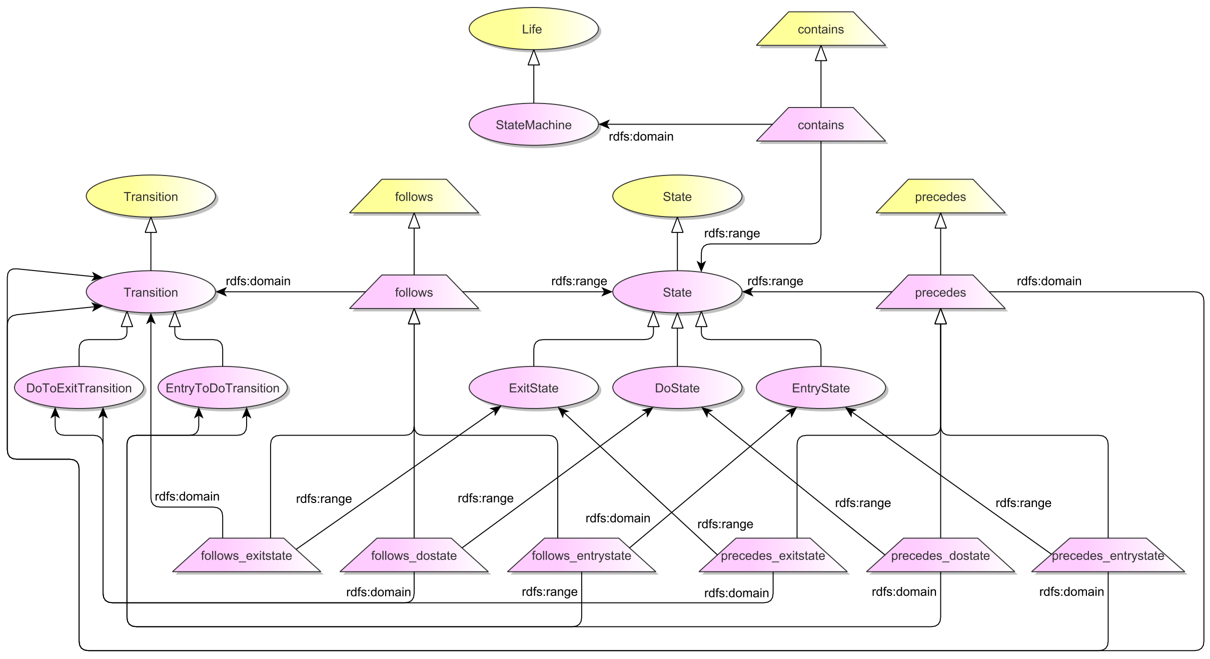


Figure 7.27: *An excerpt of the state modeling language ontology*

As the notion of state machines is widely known in the software engineering community with deep historical roots and additionally coined by the UML, the semantics of entry, do, and exit actions is unambiguous and precise for most developers. But, in context of a comprehensive software system the semantics of these ingredients must be appropriately represented. Hence, it was necessary to investigate what the shallow metamodel constructs actually mean.

In summary, the semantics of the different action types is that all actions of a type shall be executed when the underlying state machine is in a particular state-related *sub phase*. For example, all entry-actions shall be executed in the phase when a dedicated state is *reached*. Once every entry-action finished execution the *do* phase of the current state is entered and all do-actions are triggered. After a state change was initiated the state-related phase changes to *leaving*, which triggers all exist-actions.

We approached the problem by defining a particular sub state for each action type (cf. Figure 7.27). These sub states were then linked with appropriate transitions. The language ontology mapping relates entry, do, and exit actions with appropriate sub states indicating that the actions occur in corresponding state phases. Hence, Actions of a certain type can only be executed in certain system states and these states are sequentially chained. For better clarity, Figure 7.28 illustrates how the “created” state of the model in Listing 7.2 is represented in the knowledgebase.

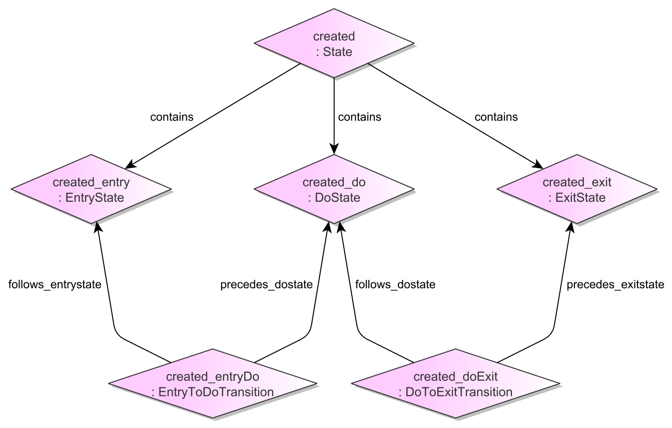


Figure 7.28: *Representing states with ontology individuals*

7.2.2.3 Action Modeling

The action modeling lanuage module provides the ability to declare invokable behavior. Some module statistics are summarized in Table 7.4.

Header	
Name	Action
ID	de.feasiple.salesscenario.action
Metamodel & Generator	
Metamodel Classes	12
Generator Target Platform	Java data type wrappers & static methods
Domain-specific Framework (classes / LOC)	5 / 260
Generator Templates (count / LOC)	1 / 68
Semantic Infrastructure	
Element Handlers	6
Attribute/Reference Handlers	–
Language Ontology Classes	7
Language Ontology Properties (incl. rev.)	3

Table 7.4: *Overview action modeling language module*

The action modeling metamodel is depicted in Figure 7.29. The DSL bundles Actions in ActionSets that are, in turn, aggregated in ActionBundless. Like in method declarations Actions can receive one or more arguments and return a particular result, which are identified with certain DataTypes. In the DSL we distinguish between SimpleTypes and SetTypess. Both are only named proxies without any further specification. Moreover, we specified typical data manipulation actions following the *CRUD (Create Retrieve Update Delete)* scheme.

In Figure 7.30 an exemplary action model is depicted to illustrate the provided generic editor enhancements. For the action modeling language module we followed a special approach when defining the code generator. As the metamodel already shows, the action modeling DSL does not cover a complete set of programming constructs, such as if statements, loops, other control structures. Although imaginable, this would have been by far too complex for a research project

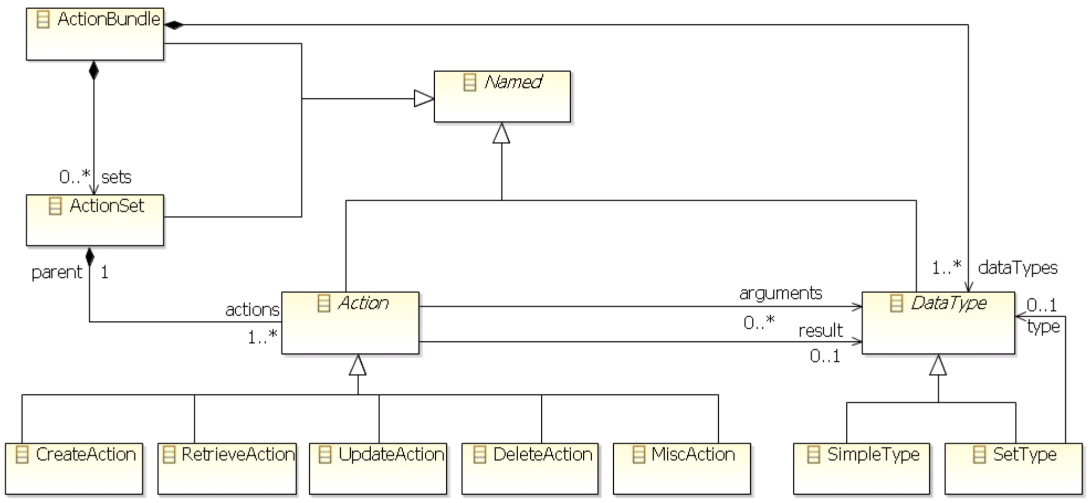
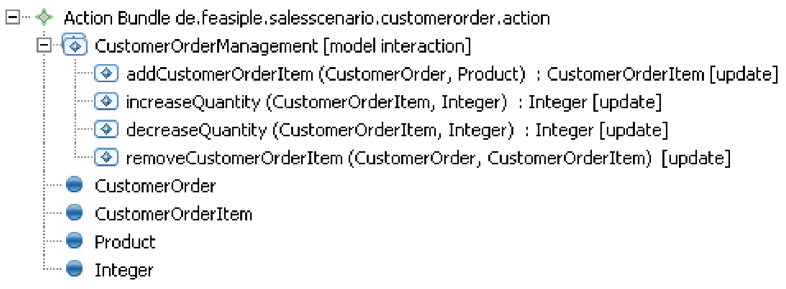


Figure 7.29: *Metamodel of the action modeling DSL*

prototype. Instead, we had to limit the DSL to prominent action types as mentioned above.

Unfortunately, sophisticated control structures were necessary to complete our case study. As a solution, we performed an outsourcing of actual business logic from model level to code level using so-called *protected regions* [EFH+08, pp. 82-83]. The openArchitectureWare generator framework allows to introduce dedicated code sections into generator templates. These code sections are then marked with unique identifiers in the output files. Within the marked sections, handwritten programming code can be specified. When (re-)executing the code generator properly marked sections are cached and regenerated when new assets are written. This way, developers are able to inject manually written code portions into generated assets.



```
Action Bundle de.feasiple.salesscenario.customerorder.action
  CustomerOrderManagement [model interaction]
    addCustomerOrderItem (CustomerOrder, Product) : CustomerOrderItem [update]
    increaseQuantity (CustomerOrderItem, Integer) : Integer [update]
    decreaseQuantity (CustomerOrderItem, Integer) : Integer [update]
    removeCustomerOrderItem (CustomerOrder, CustomerOrderItem) [update]
  CustomerOrder
  CustomerOrderItem
  Product
  Integer
```

Figure 7.30: *Exemplary action model*

Listing 7.4 shows an excerpt of the generated code from the model depicted in Figure 7.30. The code shows the `addCustomerOrderItem` action with appropriate parameter declarations and result types. The marker `ROTECTED REGION` followed the `ID` part opens the protected region while `PROTECTED REGION END` closes it. In between, arbitrary code can be provided which will last even after regeneration.

In order to achieve a proper separation between data type proxies from action models and actual data types defined with the business object DSL, we developed a small domain-specific framework. This framework provides an abstract generic data type proxy class that can be parameterized by generated subclasses. Listing 7.5 shows an excerpt of the generated proxy for the SimpleType `CustomerOrderItemParameter`.

The abstract data type proxy class is `AbstractParameter` whose type parameter is initially bound to the top level type `Object`. Later, when integrating employed language modules during

```java
package de.feasiple.salesscenario.customerorder.action;
// ...
public final class CustomerOrderManagement {
        // ...
        public static CustomerOrderItemParameter addCustomerOrderItem(
                        CustomerOrderParameter customerOrderValue,
                        ProductParameter productValue) {
                /*PROTECTED REGION ID(Custo_toolong_Item) ENABLED START*/
                // your code goes here
                /*PROTECTED REGION END*/
        }
        // ...
}
```

Listing 7.4: *Excerpt of generated code for an action model*

project engineering the type parameter can be bound to arbitrary types provided by foreign generated code. In the method definitions, wrapped objects can be queried from parameter objects and everything works well.

```java
package de.feasiple.salesscenario.customerorder.action;

import de_feasiple_salesscenario_action
        .generator.java.api.AbstractParameter;

public final class CustomerOrderItemParameter
                extends AbstractParameter<Object> {
}
```

Listing 7.5: *Excerpt of generated data type proxies for an action model*

The language ontology for the action modeling DSL is illustrated in Figure 7.31. As depicted, Actions are represented as universal Behaviors because they actually denote method declarations which are instantiated by invocation. Besides the Behavior also SimpleTypes get an ontology representation. Dedicated properties establish appropriate domain and range restrictions on top inherited USMO relations. Here, once more the semantic value of the ontological representation is elucidated. For instance, the CreateAction is appropriately marked as a Behavior which may instantiate corresponding types. This machine-readable meaning does not arise from the metamodel without any human interpretation.

A little more challenging was the integration of SetTypes. Recall from Section 6.1.1.3, that Sets are neither Universals nor Particulars but rather abstract entities that exist outside of time and space. In this respect, we could not simply map the metamodel's SetType to an USMO Universal. Instead, we had to capture that a Set *is of* a certain type without being the type's entire extension. To this end, we developed an appropriate mapping handler that correctly creates and relates individuals that involve SetTypes. For instance, Figure 7.32 shows the resulting knowledgebase indivudals for a CreateAction that returns a Set of a particular SimpleType.

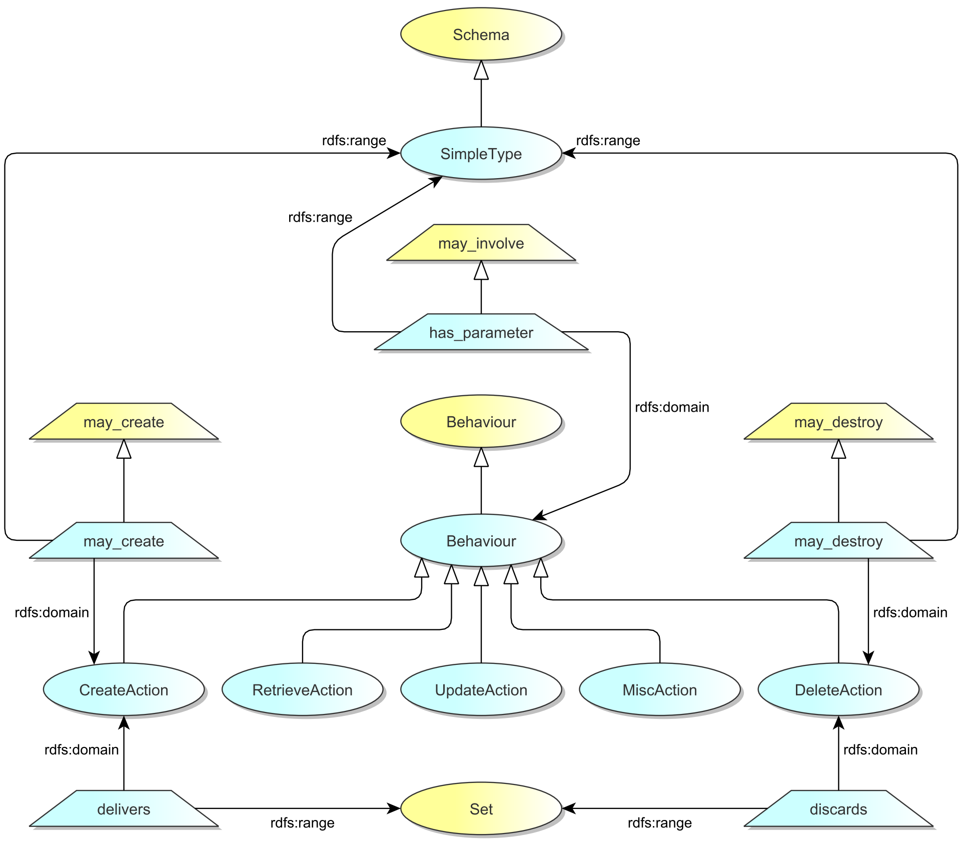


Figure 7.31: *Action modeling language ontology*

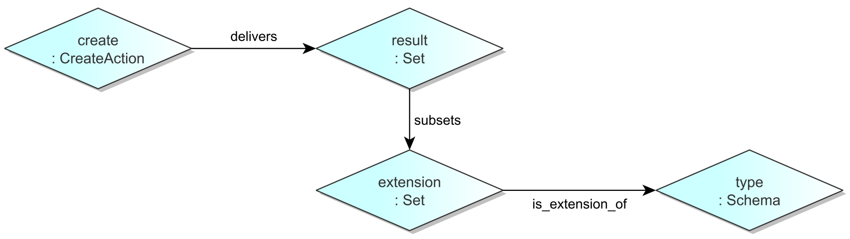


Figure 7.32: *Instantiated knowlegebase for object set creation*

7.2.2.4 User Interface Modeling

The language module that facilitates the production of user interfaces is without a doubt the most complex module that we have developed in context of this thesis. Table 7.5 gives an impression about its complexity. Here, especially the number of lines of code in the generator template but also the amount of metamodel classes foreshadows the efforts that were necessary for development.

In Figure 7.33, the metamodel of the user interface DSL is illustrated. Here, everything that is necessary to create a sophisticated UI is contained. This begins with an EventCatcher class that adds context-sensitivity to UI elements. Other significant top level classes are Dialog and Widget. Dialogs are well-known 2D windows that are decorated with a title bar and a system menu tool bar. A Dialog holds a ContainerWidget, the only widget which is allowed to hold and arrange several other widgets. To this end, a ContainerWidget may be equipped with an appropriate Layout. In addition, the metamodel provides a number of Widgets that are probably known from other UI toolkits, such as the *Standard Widget Toolkit (SWT)* [Fou, SHNM04].

Header	
Name	View
ID	de.feasiple.salesscenario.view
Metamodel & Generator	
Metamodel Classes	28
Generator Target Platform	Java + SWT
Domain-specific Framework (classes / LOC)	9 / 806
Generator Templates (count / LOC)	1 / 1443
Semantic Infrastructure	
Element Handlers	4
Attribute/Reference Handlers	1
Language Ontology Classes	6
Language Ontology Properties (incl. rev.)	5 (10)

Table 7.5: *Overview user interface modeling language module*

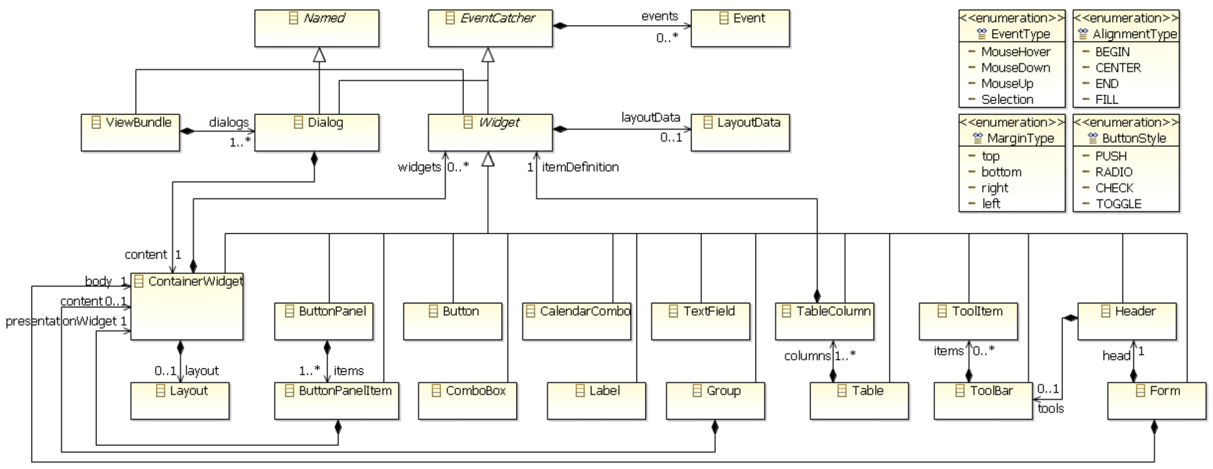


Figure 7.33: *Metamodel of the user interface modeling DSL*

Concerning the code generator for this language module we developed a huge template that produces SWT-based UIs. To this end, we created a domain-specific framework that provides base implementations for complex widgets and manages common problems in user interface development, such as parallelism between worker and UI threads. Figure 7.34 depicts an exemplary UI model that contains several of the available widgets and shows the generic editos visual enhancements for modeling. As depicted, we went beyond primitive widget containment and implemented even layouts and layout constraints for ContainerWidgets. This allows to arrange contained elements in vertical or horizontal direction, define corresponding column counts, and specify spatial extents for containers. Figure 7.35 illustrates the generated dialog for the model shown in Figure 7.34.

Compared to the metamodel, the language module's ontology is a light weight asset, as depicted in Figure 7.36. It mainly captures Widgets as the Representative EventTriggerWidget, omitting the corresponding inheritance relationship in the metamodel. In addition, a new *widget type* is identified, namely widgets that contain a caption. Such captions may be text values for labels, buttons or group boxes but also the title of a dialog, for example. Corresponding

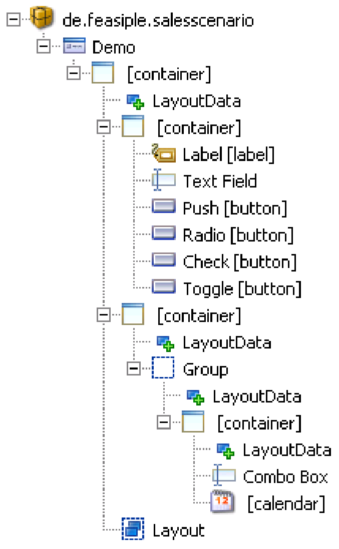


Figure 7.34: *Exemplary user interface model*

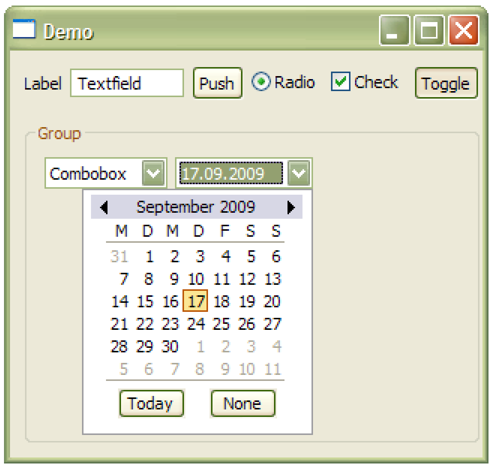


Figure 7.35: *Generated user interface for exemplary model*

CaptionWidgets contain a UICaption which is an extra Representative.

The dedicated mapping of captions makes sense because such values usually show data coming from other solution space parts, such as the data model or internationalization resource bundles. Hence, they denote a potential inter-language reference candidate. Another lanuage ontology feature is the reduction of the containment hierarchy along the widget tree to a single relation between the container dialog and its children. This is mainly justified because the reflection of the entire hierarchy into the ontology TS would not render any extra value for later integration. Moreover, this approach keeps the knowledgebase structures manageable.

The small amount of language ontology classes in the present case illustrates one key advantage of our approach. Especially for complex DSLs with many classes, properties, and constraints, the semantic architecture of HybridMDSD allows to cut out partial views that capture the core meaning and most significant relationships of the corresponding language.

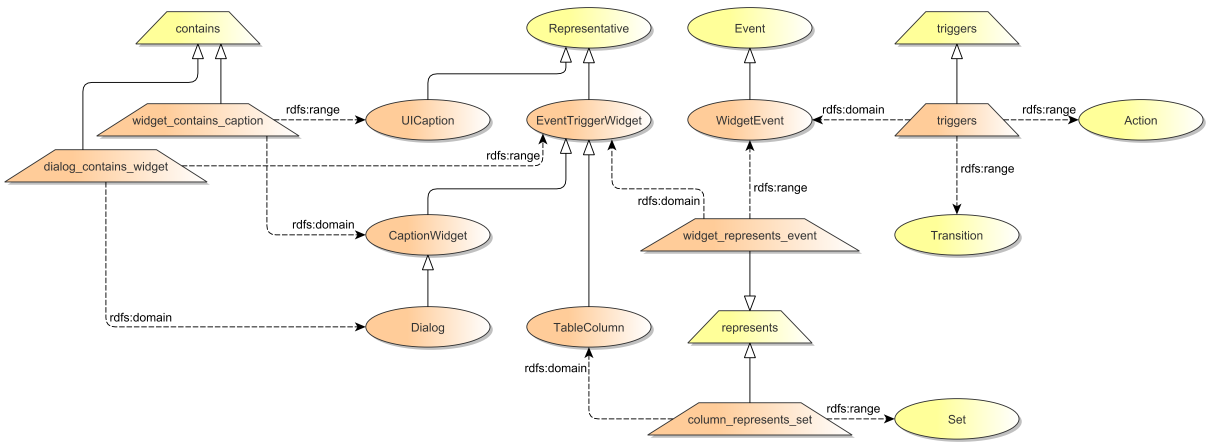


Figure 7.36: *User interface modeling language ontology*

7.2.2.5 Context Modeling

Another important language module is the context modeling module. This module is among others vital for applications that employ persistent storage and user interfaces together. It allows to buffer in-memory data models in addressable spaces. Table 7.6 gives an overview of the module and the assets it provide.

Header	
Name	Context
ID	de.feasiple.salesscenario.context
Metamodel & Generator	
Metamodel Classes	8
Generator Target Platform	Java 2 SE
Domain-specific Framework (classes / LOC)	13 / 712
Generator Templates (count / LOC)	1 / 227
Semantic Infrastructure	
Element Handlers	2
Attribute/Reference Handlers	2
Language Ontology Classes	2
Language Ontology Properties (incl. rev.)	1

Table 7.6: *Overview context modeling language module*

The concept of the context modeling language is comparable to the *session context* mechanism known from web application frameworks. Session contexts allow to store certain data objects in a server managed repository that can queried from UI templates. For example, the JBoss Seam framework maintains *seam contexts* that span corresponding data caching environments with particular validity constraints [KMR$^+$08, pp. 105-109]. In order to implement such a contextual model in form of a domain-specific language we developed the metamodel illustrated in Figure 7.37.

The metamodel design is straightforward. A ContextProvider contains available Contexts that can be organized hierarchically. Each Context contains one or more Slots, the actual representative for a data object which is to be stored later on. We distinguish between unary slots and ListSlots that contain, as the name indicates, a *list* of data objects. Moreover, a ListSlot has a particular selection, i.e., an element of the cached list that is injected into another Slot. When creating context hierarchies each context obtains a particular dependency to its parent that has an effect on context validity and lifetime. Additionally, slot values may be passedFrom other slots of parent contexts.

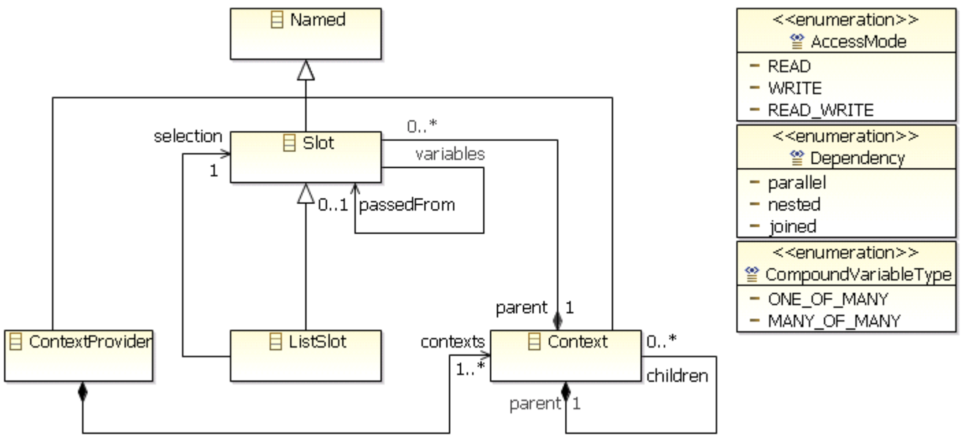
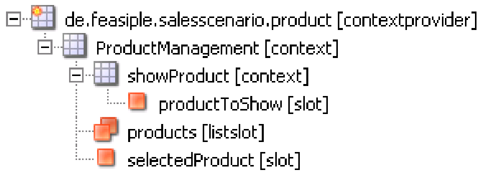


Figure 7.37: *Metamodel of the context modeling DSL*

The code generator for the context modeling language module produces assets that rely on a self-developed doman-specific framework which implements necessary semantics. To this end, mainly two challenges had to be tackled. Firstly, the slot keys needed an appropriate unique representation on code level and, secondly, appropriate code sections that fill corresponding slots with actual data were obliged to be amenable for composition operations. Listing 7.6 shows how we managed the first task. This listing shows a generated interface for the contexts depicted in the exemplary model of Figure 7.38.



```
de.feasiple.salesscenario.product [contextprovider]
  ProductManagement [context]
    showProduct [context]
      productToShow [slot]
    products [listslot]
    selectedProduct [slot]
```

Figure 7.38: *Exemplary context model*

As illustrated, the Java code generator for context models produces a cascading interface declaration that maintains corresponding slots as string constants. This way, we implemeted a solution space-wide type safe way to address and use slot keys. For example, the productToShow slot would be addressed using the `ProductManagement.ShowProduct.PRODUCTTOSHOW` constant[1].

We solved the second challenge equally straightforward, as shown in Listing 7.7. Thus, the code generator produces for each context a *creator* class that manages slot instantiation in dedicated methods. Here, each slot is treated in a separate method which paves the way for later integration with, e.g., the action modeling module to trigger data retrieval from persistent storage. Each context creator implements the framework interface `IContextCreator` and is managed by the generated `ContextProvider`.

[1] For brevity reasons we omitted the leading package name.

```java
package de.feasiple.salesscenario.product;

import de_feasiple_salesscenario_context.generator.java.api.Contexts;

public interface ProductManagement extends Contexts {

	interface ShowProduct extends ProductManagement {
		String PRODUCTTOSHOW = "productToShow";
	}

	String PRODUCTS_LIST = "products";

	String SELECTEDPRODUCT = "selectedProduct";
}
```

Listing 7.6: *Generated context interface*

```java
package de.feasiple.salesscenario.product;
// ...
public class ProductManagementContextCreator implements IContextCreator {
	// ...
	@Override
	public Context createContext(Context parent,
			Class<? extends Contexts> key) {
		current = parent.createSubContext(key);
		ISlot<?> slot;
		slot = this.createProducts(parent);
		slot.enforceAccessMode();
		current.addSlot(slot);
		// ...
		return current;
	}

	public ListSlot createProducts(Context parent) {
		ListSlot slot = new ListSlot("products");
		slot.setAccessMode(AccessMode.READ);

		// TODO: implement value creation here
		return slot;
	}
	// ...
}
```

Listing 7.7: *Generated context creator*

While the efforts to develop the context modeling code generator were significant the language ontology is (like the metamodel) rather tiny, as illustrated in Figure 7.39. The most important classes Context, ListSlot, and Slot receive a semantic representation. To this end, Context and ListSlot both denote Sets because they aggregate Substantials, i.e., the data objects represented by Slots. Set membership between Slots and ListSlots is asserted with the selection property.

Moreover, we created an extra reference handler that observes the instantiation of passedFrom relations. The handler properly sets an `owl:sameAs` relation between the passing and the receiving slot to signal semantic equivalence. Figure 7.40 illustrates the resulting knowledgebase for the model shown above.

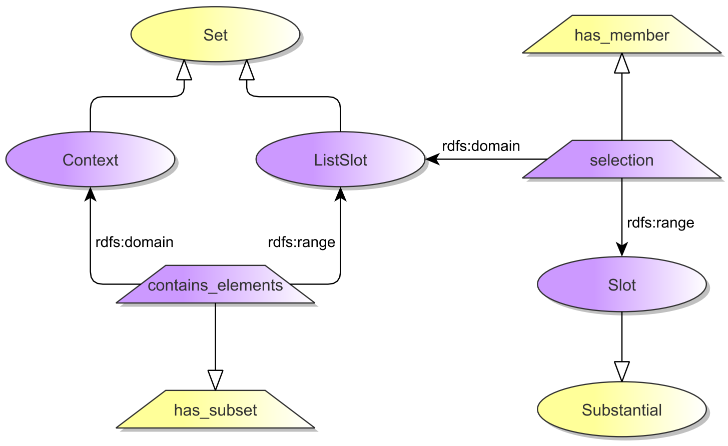


Figure 7.39: *Context modeling language ontology*

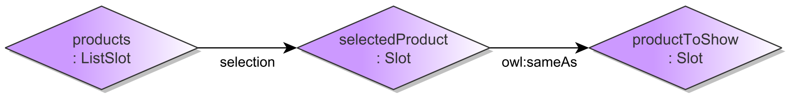


Figure 7.40: *Knowledgebase representation of exemplary context model*

7.2.2.6 Pageflow Modeling

The last language module that we introduce in this section is a module for pageflow modeling. Table 7.7 gives an overview. As the table lists, the module is named "dialog" instead of "pageflow". This has historical reasons. The DSL classification in Section 6.1.1.2 revealed a language class called "dialog languages". Succeedingly, when elaborating technologies for the implementation of this case study we revealed the JBoss Seam framework which involves jPDL *pageflows* [KMR$^+$08, pp. 181-196], a more appropriate naming in our opinion.

Header	
Name	Dialog
ID	de.feasiple.salesscenario.dialog
Metamodel & Generator	
Metamodel Classes	9
Generator Target Platform	Java + SCXML
Domain-specific Framework (classes / LOC)	–
Generator Templates (count / LOC)	1 / 151
Semantic Infrastructure	
Element Handlers	1
Attribute/Reference Handlers	–
Language Ontology Classes	5
Language Ontology Properties (incl. rev.)	2

Table 7.7: *Overview pageflow modeling language module*

The pageflow DSL's metamodel is rather simple and very narrow to the jPDL pageflow language of JBoss Seam (cf. Figure 7.41). A grouping PageFlowContainer holds all PageFlows in a model. A PageFlow contains diverse Nodes which might be usual Page nodes or Decision nodes that encapsulate a boolean condition which is to be evaluated during runtime. Depending on the evaluation result on of two possible outcome transitions is chosen. Figure 7.42 illustrates an exemplary pageflow model that shows domain-specific editor enhancements.

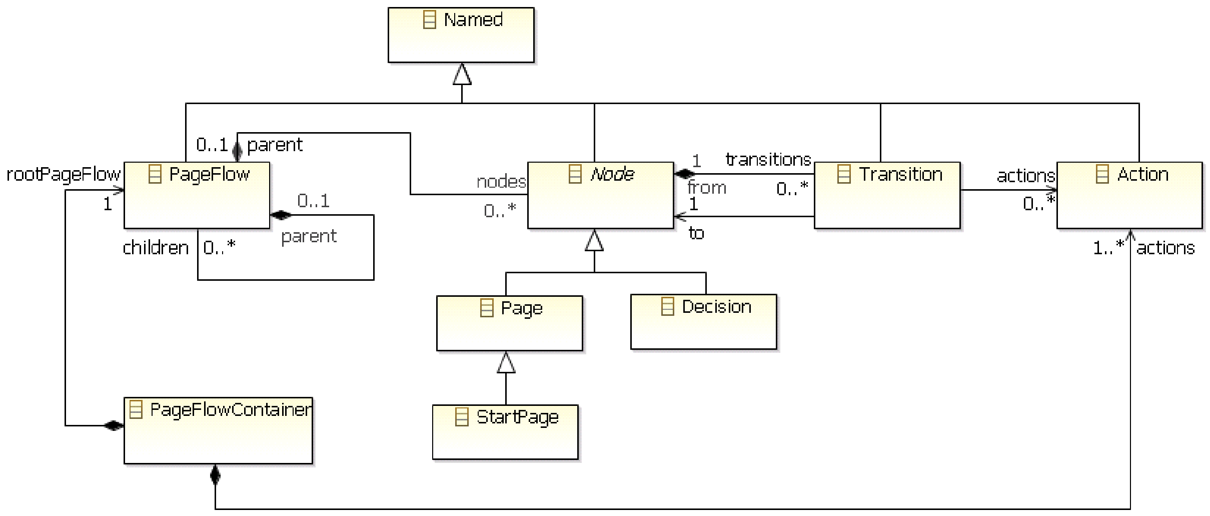
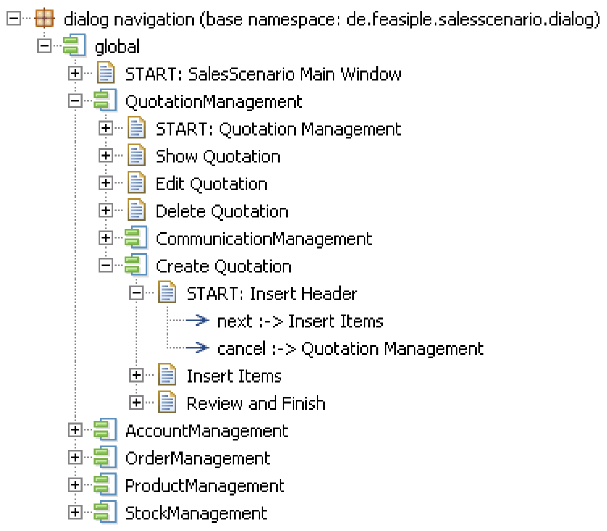


Figure 7.41: *Metamodel of the pageflow modeling DSL*

The language ontology of the pageflow modeling language does not hide any secrets. A rather direct mapping from metamodel classes to ontology classes coins the ontology. One special feature is- the explicit dependency between Decisions and particular TemporalQualitys. This is due to the fact that corresponding boolean expressions usually involve volatile context parameters, which need to be represented properly. To ensure that a PageFlow contains only one StartPage we created an appropriate integrity rule because OWL does not support qualified cardinality restrictions.



```
dialog navigation (base namespace: de.feasiple.salesscenario.dialog)
  global
    START: SalesScenario Main Window
    QuotationManagement
      START: Quotation Management
      Show Quotation
      Edit Quotation
      Delete Quotation
      CommunicationManagement
      Create Quotation
        START: Insert Header
          next :-> Insert Items
          cancel :-> Quotation Management
        Insert Items
        Review and Finish
    AccountManagement
    OrderManagement
    ProductManagement
    StockManagement
```

Figure 7.42: *Exemplary pageflow model*

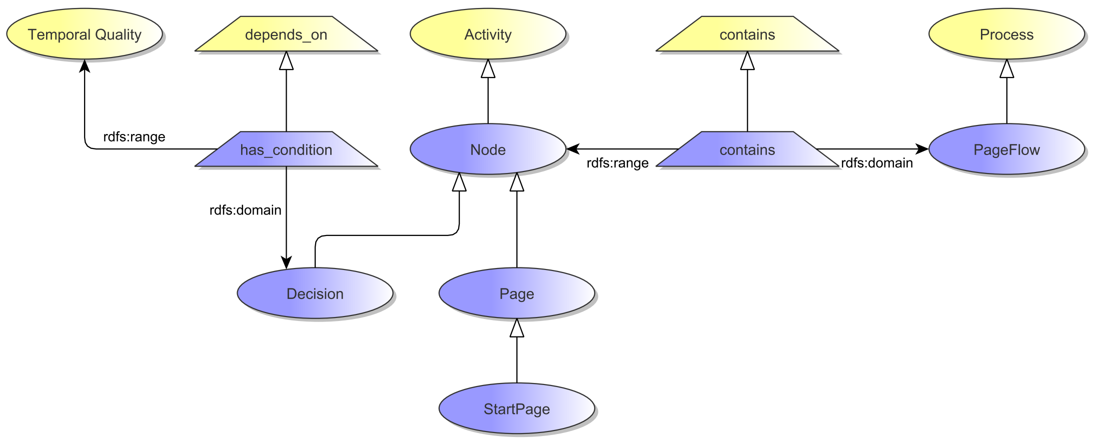


Figure 7.43: *Pageflow modeling language ontology*

7.2.3 Project Engineering: Integration

Because all language ontologies are derived from the central upper ontology USMO the semantic integration task is already (at least partially) accomplished. The information gained from the mapping between modeling constructs and USMO classes qualifies the mapped constructs in terms of these classes and allows to establish predefined relationships. In a project-specific context, however, it might be useful to provide an alternative terminology for these connections. To this end, we subclass particular USMO properties in the project ontology, assign an appropriate name, and adjust domain and range definitions to connecting language ontology types. The pros and cons of this approach are discussed in Section 7.4.

In the following we shed light on selected examples for the integration of introduced language modules. For each example we (1) explain how involved semantic representations are integrated and (2) how composition generators implement code composition.

7.2.3.1 Data and States

For the integration of data types (i.e. Schemas) and state machines (i.e. Lifes), USMO defines the property may_live. Thus, we used this property as basis for the more appropriately named relation has_lifecycle as illustrated in Figure 7.44.

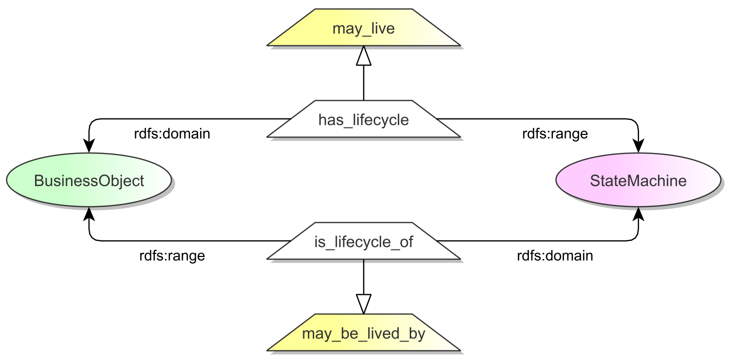


Figure 7.44: *Semantic integration of business objects and state machines*

The integration on code level was equally straightforward. As introduced in Section 7.2.2, both language modules' generators produce Java classes. The business object generator yields plain POJOS without any specific business logic and the state machine generator produces a facade class that allows to trigger transition events. The semantics of the inherited may_live property defines that each Substantial that instantiates a BusinessObject has or may have a Life which is defined by a StateMachine. On code level this means for each runtime instance of the Java class representing a BusinessObject model instance a runtime instance of the Java class that represents a StateMachine model instance must be assigned. Listing 7.8 shows the desired composition result.

```java
package de.feasiple.salesscenario.quotation;
// ...
public class Quotation {
        // ...
        private String title;

        // composition result - begin
        private QuotationLifeCycle lifeCycle = new QuotationLifeCycle();

        public String getState() {
                return lifeCycle.getCurrentState().toString();
        }
        // composition result - end
}
```

Listing 7.8: *Excerpt of composed generated code integrating data and behavior assets*

To get an impression of the composition generator that implements this composition and the integration code that is generated, the reader is referred to the Listings 6.2 and 6.3 on page 108 and 109 respectively. For technical reasons also a relation from StateMachine back to BusinessObject needs to be introduced, as shown the listing.

7.2.3.2 UI and Contexts

User interface components usually illustrate data from the persistence layer. This implies that the data is provided to the UI implementation which is realized with the context modeling module. In order to fix the necessary integration between UI and contexts, we decided to assign one Context per Dialog. The corresponding connection on semantic level is shown in Figure 7.45.

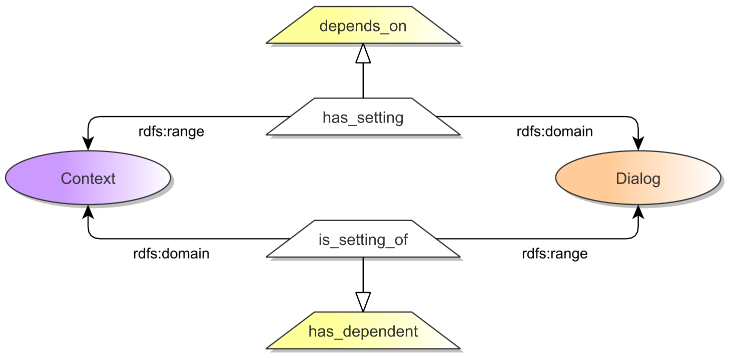


Figure 7.45: *Semantic integration of UI dialogs and contexts*

The property has_setting derives from the USMO base dependency relation. On code level, the implementing class for Dialog needs to have access to the referenced Context. We implemented this with the injection of the relevant context into a key-value based property map of the underlying UI framework SWT, which is available in the Dialog class. Listing 7.9 shows the composition code that is produced by the composition generator for a particular inter-model reference. The aspect injects illustrated lines of code into the constructor of the Dialog class. The corresponding connection relates the confirmation dialog for a deletion operation with the necessary context `SalesScenario.QuotationManagement.DeleteQuotation`.

```
package de.feasiple.salesscenario.quotation.view_context.deletequotation;
// ...
public aspect WeaveContextIntoDeleteDialog {
        // weave context to dialog
        after(DeleteQuotationDialog dialog) returning: this(dialog)
                && initialization(DeleteQuotationDialog.new(..)) {
                dialog.getControl().setData(ContextProvider.INSTANCE
                                .getContext(
                                        SalesScenario
                                                .QuotationManagement
                                                .DeleteQuotation.class));
                dialog.update();
        }
}
```

Listing 7.9: *One aspectual composer for the integration of UI and context models*

7.2.3.3 Data and Contexts

In Section 7.2.2.5, we presented the context modeling language. As explained, this DSL does not provide any means to specify *which* kind of data is managed but rather allows to implement *how* certain data is arranged in uniquely named settings. Hence, for an integrated scenario it was necessary to bind particular type declarations to corresonding Slots in order to achieve type safety across language modules. Figure 7.46 depicts our solution.

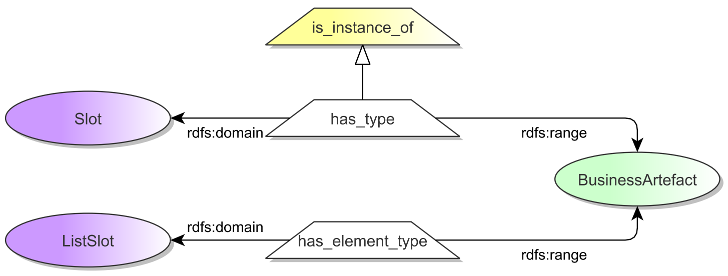


Figure 7.46: *Semantic integration of business objects and UI contexts*

As Slots inherit from USMO Substantial, they are intensionally defined by some Schema. These, in turn, are available from the data modeling DSL in form of BusinessArtefacts. Thus, we defined the properly named relation has_type between both which inherits from is_instance_of. Relating ListSlots with a dedicated type was a little more problematic. Recall from Section 7.2.2.5, that the ListSlot concept inherits from USMO's Set. Since a Set contains only Particulars and is itself an independent USMO top level type, no appropriate predefined property can be used to relate ListSlots with any universal type definition.

As a solution we defined the property has_element_type that facilitates to create relations between ListSlots and BusinessArtefacts. Moreover, we created the following integrity rule that restricts ListSlot sets to contain only BusinessArtefacts of the same type.

$$ListSlot(x), BusinessArtefact(y), has\_element\_type(x, y), has\_member(x, z) \tag{7.1}$$
$$\rightarrow is\_instance\_of(z, y) \tag{7.2}$$

In context of our prototypical implementation that provides an ontology adapter for the semantic web framework Jena (cf. Section 7.1.2.1), we implemented this rule amenable for the Jena *general purpose rule engine*. The result is shown in Listing 7.10.

```
[ensureListSlotTypes:
  (?listslot rdf:type context:ListSlot),
  (?type rdf:type businessobject:BusinessArtefact),
  (?listslot context:has_element_type ?type),
  (?listslot usmo:has_member ?e) ->
    (?e rdf:type ?type)]
```

Listing 7.10: *Jena integrity rule to ensure type safety between language modules*

Interestingly, an integration on code level was not necessary for this inter-language connection. Instead, the semantically ensured type safety operates in combination with other language modules as explained below.

7.2.3.4 Pageflows and UI

Another important interface exists between the pageflow definition language and the user interface DSL. Since pageflows define navigation paths through the UI corresponding Pages need to obtain a particular counterpart — the Dialog. Figure 7.47 illustrates the relevant excerpt of the project ontology.

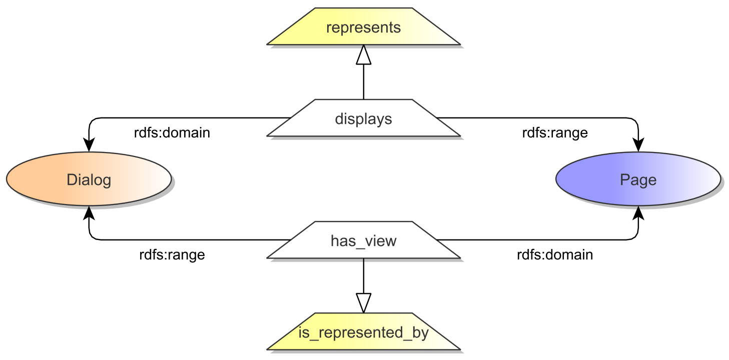


Figure 7.47: *Semantic integration of pageflows and UI dialogs*

As depicted, we established the properties displays and has_view along the representation relationship of USMO. Thus, each pageflow Page gets a representation as Dialog. In this respect, the more difficult part was the integration on code level. This is because the actual semantics of the pageflow language needed to be realized, i.e., each time a Page is reached the correspondingly connected Dialog must be opened.

We analyzed the generated code of the pageflow DSL and quickly found the connection point. Like the state modeling generator also the pageflow generator produces an SCXML-based state machine facade that allows to integrate encapsulated behavior. The generated facade contains a method for each Page in the modeled pageflow. In case a Page is reached because a particular Transition was triggered before, the corresponding method is called by the SCXML framework.

```
package de.feasiple.salesscenario.quotation.view_dialog;
// ...
public aspect WeaveQuotationManagement {
        // ...
        pointcut showQuotationCall() :
                execution(* de.feasiple.salesscenario.dialog.Global
                        .global_quotationManagement_showQuotation());

        void around() : showQuotationCall() {
                DialogHandler.getInstance()
                        .openDialog(ShowQuotationDialog.class);
        }
        // ...
}
```

Listing 7.11: *One aspectual composer for the integration of pageflow and UI models*

Hence, a composition generator was necessary that injects appropriate framework calls which opened the connected Dialog. Listing 7.11 shows one exemplary composition aspect produced by this composition generator.

The generated aspect defines a pointcut that attaches to the method which represents the showQuotation page, namely `global_quotationManagement_showQuotation()` in the generated pageflow facade of the application global pageflow. Within this method the domain-specific framework of the user interface module is invoked that handles opening and closing of application wide dialogs.

7.2.3.5 UI, Actions, Data, and Contexts

As last integration example we want to shed light on a more sophisticated relation between *four* different language modules at a time. Thus, we consider a *complex relation* according to the relation types introduced in Section 6.3.4.3. To get a better introduction to the use case we start with an illustrative knowledgebase excerpt (see Figure 7.48).

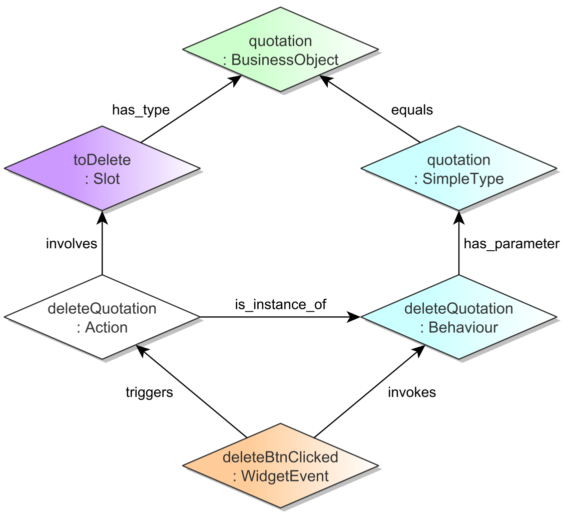


Figure 7.48: *Knowledgebase excerpt for complex relation*

The considered scenario involves the language modules View, Action, Businessobject, and Context. More precisely, it covers the UI-triggered invocation of a particular data modification action where the necessary data instances are provided by a properly typed context slot. Hence, we consider the typical case that a software user clicks on a user interface button which entails the manipulation of displayed data.

Figure 7.48 shows involved individuals. The figure's bottom shows the triggering Event that is linked through the project ontology property invokes with the deleteQuotation Behavior that shall be called. This in turn is defined to receive a parameter of the SimpleType quotation which is equals to its root BusinessObject declaration. The action call is represented by the deleteQuotation Action which receives the actual parameter instance, held in the surrounding context Slot toDelete. Please note that we have omitted this Context individual in Figure 7.48 for the sake of clarity.

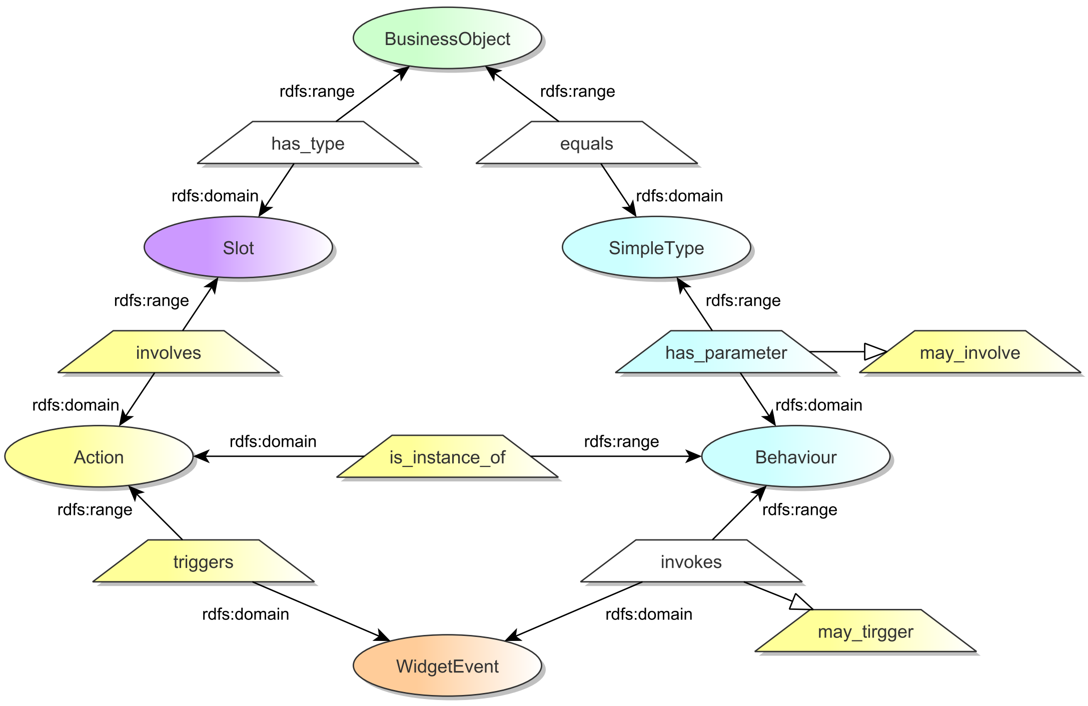


Figure 7.49: *Semantic integration of four language modules*

Figure 7.49 shows how the different language ontologies are integrated. Firstly, the UI DSL may invoke Behavior declarations of the action DSL. Secondly, action DSL's SimpleTypes are related to BusinessObjects of the data DSL through the equals property. To this end, we additionally provided the Jena inference rule shown in Listing 7.14 that yields semantic equivalence for OWL DL reasoners. Thirdly, the involved Slot is *typed* with the appropriate BusinessObject using the has_type relation.

```
[deriveEquivalence:
  (?e1 hmdsd:equals ?e2) ->
    (?e1 owl:sameAs ?e2)]
```

Listing 7.12: *Jena inference rule to derive semantic equivalence*

In addition, we want to draw dedicated attention to the relation between WidgetEvent, Slot, and Behavior. This complex connection is established through the linking Action class, since the event triggers the action that is_instance_of the corresponding behavior. To establish such connections appropriate inference rules must be provided that persistently found corresponding structures and relate it to the original relation. For the present case, for instance, the actual call represented by the deleteQuotation Action must be emitted in case a WidgetEvent is related to some Behavior using the invokes relationship.

Like the semantic connection, also the code composition pattern is more complex than in the use cases shown above. Listing 7.13 gives an insight into its complexity. The listing illustrates the oAW template-based composition generator for the considered [n:m] relation (cf. Section 6.2.2.3). As depicted, the composition generator imports all necessary DSLs and is instantiated with all involved relation piers. Moreover, the template's header section imports the generator profile's of the module's to integrate. These are used within the template body. For example, the `id()` calls with corresponding arguments return full-qualified type or method names that particular generators produce for passed model elements. Another example is the `methodCall()` invocation that returns a Java method call statement for the passed Action and argument list.

```
«IMPORT de_feasiple_salesscenario_action»
«IMPORT de_feasiple_salesscenario_context»
«IMPORT de_feasiple_salesscenario_businessobject»
«IMPORT de_feasiple_salesscenario_view»

«EXTENSION de_feasiple_salesscenario_action::generator::java::profile»
«EXTENSION de_feasiple_salesscenario_context::generator::java::profile»
«EXTENSION de_feasiple_salesscenario_businessobject::generator::java::prf»
«EXTENSION de_feasiple_salesscenario_view::generator::swt::profile»

«EXTENSION de::feasiple::salesscenario::hmdsd::util::ext::util»

«DEFINE aspect(Dialog dialog, Event event, BusinessObject bo,
        Slot slot, Action action) FOR Void-»
«FILE getAspectFile(dialog, event, bo, slot, action)-»
package «getPackage(dialog, event, bo, slot, action)-»;
// ...
public aspect invokes_Dialog_Event_BusinessObject_Slot_Action {

        void around(«id(dialog)-» dialog) :
                execution (void «id(event)-»)
                && this(dialog) {
                IContext context = (IContext) dialog
                        .getControl().getData();

                «id(bo)-» «argument(bo)-» = («id(bo)-») context
                        .getBinding(«id(slot)-»);
                «methodCall(action, {}.toList().add(argument(bo)))-»

                proceed(dialog);
        }
}
«ENDFILE»
«ENDDEFINE»
```

Listing 7.13: *Composition generator for an [n:m] relation between UI*

Finally, Listing 7.14 shows the output produced by above shown composition generator for one particular inter-model reference. The generated aspect's content reveals which connection is implemented. Thus, the corresponding solution space (see next section) contains an action model that defines a particular deleteQuotation action that awaits a Quotation to delete.

```
package de.feasiple.salesscenario.hmdsd.invokes;
// ...
public aspect WeaveDeleteQuotationCall {

        void around(DeleteQuotationDialog dialog) :
                execution (void de.feasiple.salesscenario.quotation.view
                        .DeleteQuotationComposite
                        .buttonDeleteOkMouseDown(..))
                && this(dialog) {
                IContext context = (IContext) dialog.getControl()
                        .getData();

                Quotation quotation = (Quotation) context
                        .getBinding(SalesScenario.QuotationManagement
                                .DeleteQuotation.DELETEDQUOTATION);

                QuotationManagement.deleteQuotation(
                        new QuotationParameter(quotation));
                proceed(dialog);
        }
}
```

Listing 7.14: *Aspectual composer for the integration of UI*

The invocation of this action is triggered with a button in an appropriate dialog that asks for deletion confirmation. The actual data instance to delete is held in the dialogs context `SalesScenario.QuotationManagement.DeleteQuotation`. Please note that we have exchanged full-qualified with local type names in the listing for the sake of clarity.

7.2.4 Modeling

The last part in HybridMDSD development is the actual modeling phase where domain specialists can finally take advantage of previously invested efforts. As explained in Section 6.3.5, this phase is divided into (1) problem space modeling, (2) solution space modeling, and (3) the creation of a mapping in between. In this section, we will briefly present (1) with the feature model for the SalesScenario case study and (2) with an excerpt of the overall project knowledge-base instantiated by corresponding mapping handlers based on created solution space models. Because (3) is out of scope in this thesis we will mention the mapping between problem and solution space only when necessary.

7.2.4.1 Problem Space Modeling

Main purpose of the Sales Scenario is the holistic management of business data, including central storage and access controlled retrieval. It focuses on product sales processes, where the core features comprise: *Customer Order Management*, *Payment*, *Account Management*, *Product Management*, and *Communication*. These features are depicted in the case study's feature model in Figure 7.50, created with an Eclipse-based feature modeling tool developed by Christian Wende et al. from Dresden University of Technology, Germany. The model's visual syntax relates to the original FODA notation (cf. Section 2.3.2.1) but is purely cardinality-based and abstains from the original, often confusing bubble syntax.

In addition to the core features we derived further variation points that are more or less crosscutting. The *Communication* feature, for instance, crosscuts massively as also indicated in the case study's architecture overview (cf. Figure 7.22 on page 155). This is because various steps during sales processing involve customer interaction. In such cases, the SalesScenario queries the user for the desired communication option according to the features that are available in the corresponding variant.

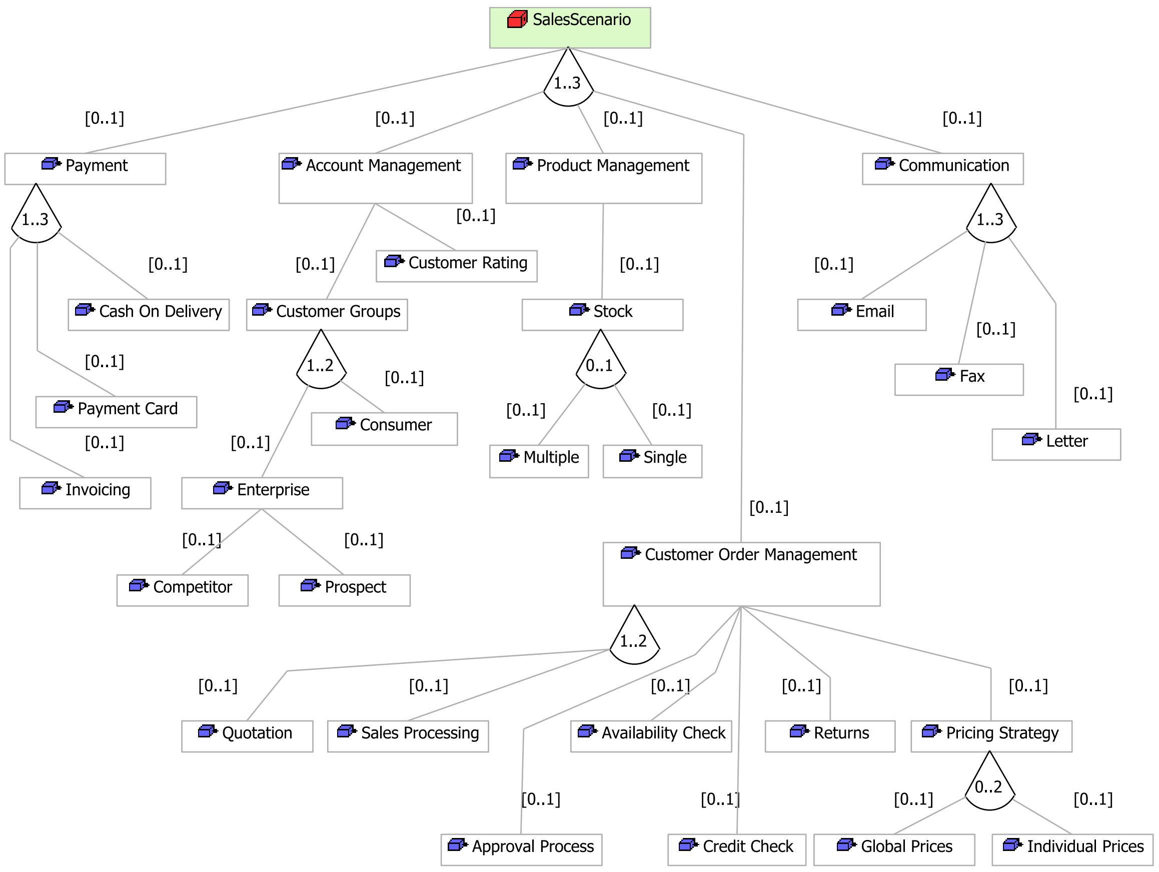


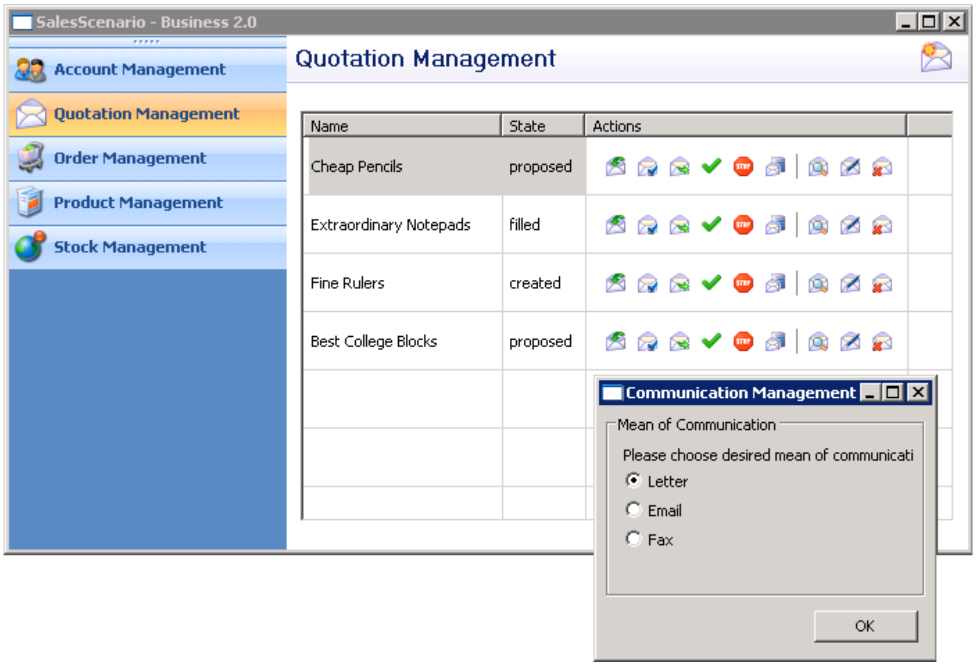
Figure 7.50: *Feature Model of the SalesScenario product line (from [LG08b])*

7.2.4.2 Solution Space Modeling

When introducing the language modules that were developed for the SalesScenario in Section 7.2.2, we have always shown example models to illustrate the visual syntax of corresponding DSLs. The illustrated models nearly all together stem from the SalesScenario solution space. This is why we desist from the illustration of additional solution space models in this section. Instead, Figure 7.52 shows an excerpt of the SalesScenario project knowledgebase.

In detail, we extracted the example which was already addressed in the previous section, namely the integration of the *Communication* feature. The setting is as follows. When a product quotation is proposed to selected customers, an appropriate dialog appears that offers available communication options. This is because the quotation is to be proposed effectively. The options to choose depend on the features that are integrated in the actual product implementation.

Figure 7.51 shows the main window of a generated SalesScenario variant. In front of the main window the communication choice dialog is shown. Recall from the feature model in Figure 7.50 that available options are *Email*, *Fax*, and *Letter*. Since all these features are contained in the depicted variant all options are rendered in the confirmation dialog.



Name	State	Actions
Cheap Pencils	proposed	
Extraordinary Notepads	filled	
Fine Rulers	created	
Best College Blocks	proposed	

Mean of Communication

Please choose desired mean of communicati

[✓ Letter] Email Fax

Figure 7.51: *Main window of the generated SalesScenario*

The corresponding knowledgebase excerpt that maps this scenario is depicted in Figure 7.52. The figure is to be read from bottom to top. The reddish individuals are emitted from the UI module and comprise a representation for the confirmation dialog (selectCommDlg) and contained ok and cancel buttons (okBtn and cancelBtn). The blueish individuals stem from the pageflow module and represent the navigation path from the quotation management dialog (quotManagement) to the communication confirmation dialog and back, including appropriate Transitions.

The former path is triggered by a click on the propose button of a particular quotation which causes the proposeClicked event to be emitted. The latter path is triggered by the cancelClicked event coming from the communication dialog. In case the communication dialog's ok button is clicked, not only a transition back to the quotation management dialog is triggered but also a change in the lifecycle of the underlying quotation is caused. To this end, the ready_propose transition is triggered that entails the state change of the quotation's lifecycle from ready to propose. The actual quotation instance who's lifecycle changes here is provided by the context slot selectedQuot which points to the currently selected quotation in the table shown in the quotation management dialog. The whole set that is listed their is delivered by the previously executed queryQuotations action.

The considered excerpt and the entire SalesScenario project knowledgebase represents a static snapshot of the dynamic semantics of the underlying software system. This facilitates the detection of design flaws and lacks in the application's control and data flow to a certain extent. The next section gives an impression about these and other features provided by the HybidMDSD architecture.

7.3 Application of the HybridMDSD Architecture

In the preceding sections we comprehensively presented the prototypical implementation of the HybridMDSD approach as well as an industrial-scale case study which examined the quality of

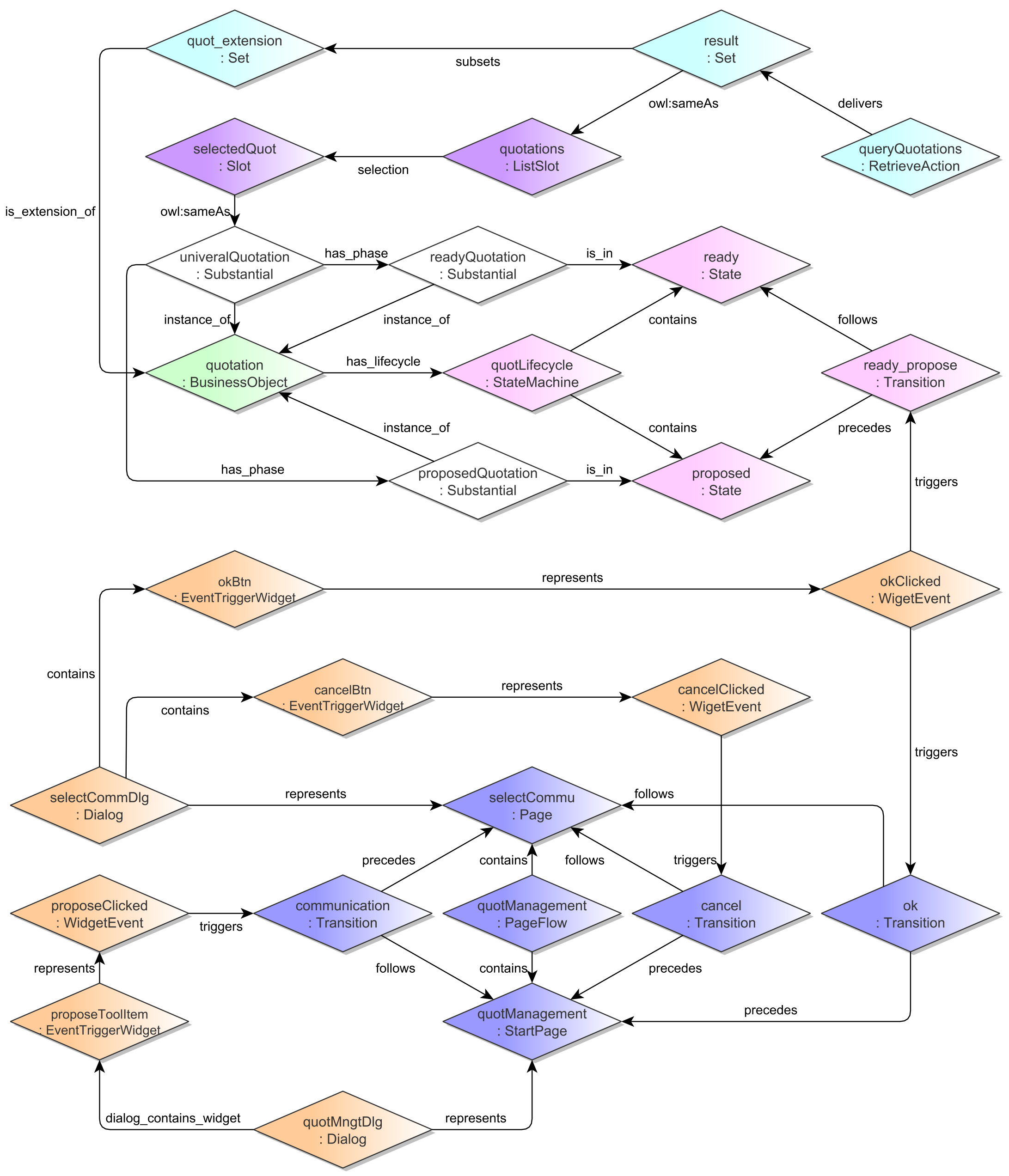


Figure 7.52: *Excerpt of the SalesScenario project knowledgebase*

both the approach and the prototype respectively. Even though the previous sections have already emphasized the power of multi-domain engineering with semantic foundations, this section is dedicated to present selected advantages that are prominent and that we have experienced during development.

We distinguished the section into the presentation of the advantages of consistency ensurance (see next section), modeling guidance (see Section 7.3.2, and the generation of integration structures on code level (see Section 7.3.3).

7.3.1 Preserving Consistency

One of the outstanding advantages of Semantic Web technologies is that they provide sophisticated means to apply logical deduction. Different kinds of logical constructs allow to check for consistency and derive additional information of asserted facts (cf. Section 2.4.3). In context of HybridMDSD this is especially useful when creating references between actually unrelated DSLs. It allows to establish a powerful basis for the detection of design flaws and incomplete specifications, as already mentioned in Section 6.3.5.2. Moreover, in product line engineering scenarios it allows to detect imperfect feature mappings which arise during sequential mapping development and are common in complex projects, as illustrated in Figure 6.38 on page 131. In the following we present a brief example which shall illustrate the nature of applied reasoning power.

Recall the example from Section 7.2.3.5 where we demonstrated the integration of multiple language modules at a time. The example presented the invocation of a service definition from the action DSL, invoked by a widget event modeled in the UI DSL, with a parameter instance object coming from the context DSL, and with type-safety given by the data DSL. Corresponding inter model references were drawn using our Eclipse-based tool suite. Now, the semantic foundation allows us to detect errors in corresponding references. For example, consider the following integrity rule:

$$\mathsf{involves}(a, s), \mathsf{is\_instance\_of}(a, b), \mathsf{is\_instance\_of}(s, sc_1), \rightarrow \exists sc_2(\mathsf{may\_involve}(b, sc_2) \wedge s_1 \doteq sc_2) \quad (7.3)$$

It ensures that, if an Action a involves a Substantial s and a is instance of some Behavior b than there must be some Schema that may_be_involved by b and that is type of s. In other words, this rule ensures that a service definition specified with the action DSL receives correctly typed instances. As we implemented our prototype on the basis of the Jena semnatic web framework, we implemented the rule-based axiomatization of our upper ontology in a format amenable to the Jena built-in inference engine. The Jena rule for the above stated constraint, thus, looks as presented in Listing 7.15.

```
[validateBehaviorTypeSafety:
   (?a usmo:involves ?s),
   (?a usmo:is_instance_of ?b),
   (?s usmo:is_instance_of ?sc1),
   noValue(?b usmo:may_involve ?sc1) ->
      (?a rdf:type sc:Invalid),
      (?a sc:caused_by ?s),
      (?a sc:violates 'Action involves instance of wrong or undefined type')]
```

Listing 7.15: *Jena integrity to ensure type-safety*

The integrity rule illustrates how we apply the semantic connector utility ontology (cf. Section 7.1.2.2) to project constraint violations into a detectable format. We classify violating constructs as `Invalid` and provide a brief explanation behind the `violates` property. This information is then adequately presented in form of Eclipse error markers that furnish mapped models and model elements.

7.3.2 Guiding the Modeling Process

Besides detecting integrity violations HybridMDSD's semantic foundation allows to realize more complex guidance scenarios. Here, we distinguish mainly two types of guidance that are explained below. In addition, we comment on guidance limitations within HybridMDSD.

Active Creation Assistance

During the modeling process solution space specifications are created step by step. Hence, models that are not completely finished exhibit inconsistencies which can be detected and handled. In addition to DSL-based constraints that might be provided by the corresponding model editor, the ontology konwledgebase that partially represents modeled content allows to provide help for the creation of incomplete models. Moreover, such inconsistencies can be managed across domain borders. Thus, even if all domain-specific constraints are satisfied it might be that particular inter-model references are necessary in order to finally derive a working software system.

To this end, three major tasks need to be performed. First, a *consistency constraint* is to be specified that must hold for the solution space. Such constraints may be DL-based or rule-based and may be provided by fixed architecture assets, such as the axiomatization of USMO, or optional domain-specific assets provided by language modules. Second, a *creation recipe* must be specified that fixes the detected inconsistency. This reaches from simple instructions to fulfill cardinality restrictions to invocations of complex adaptation rules. Last, a *framework component* is necessary that (1) detects the inconsistent state based on the defined constraint, (2) proposes possible creation actions based on the defined recipe, and (3) executes the creation action when desired by the modeler. Appropriate tooling allows to present creation assistance hints in the Eclipse workbench.

Model Repair Assistance

Besides guidance during model creation, especially product line scenarios involve additional use cases. Recall from Section 2.3.3.2 and 6.3.5 that the mapping of features to elements in the solution space is applied during product instantiation. Here, a subset of the product line feature model prescribes which solution elements will belong to the final product. To this end, initial specifications are modified. Model elements are removed or added and attribute values are adjusted.

The feature-driven modification of solution space models may result in inconsistent system states in case feature mappings are erroneous or several mappings in combination do not harmonize. Here, the model repair assistance feature of HybridMDSD comes into play. The basic idea is to define automatic *repair actions* for domain-specific languages and use cases. To this end, corresponding language modules need to provide appropriate adaptation rules that handle dedicated inconsistency states. For better illustration consider the following example.

Recall from Section 7.2.2.6 that we developed a pageflow modeling language which allows to specify navigation paths through an application's user interface. We assume that a domain-specific requirement is that the software's pageflows must not contain any gaps between its starting and ending pages. Thus, in case a feature-driven modification to a corresponding model results in a broken pageflow, an appropriate repair action should fix this gap and indicate the change accordingly.

To realize this scenario we need to define a number of rules. The overall application pattern is equal to the one followed with creation assistance above. The only difference is that the we define the last step, i.e., the decision whether a rule shall be executed or not, as optional. This way language module developers are allowed to define full automatic adaptation without obligation to consult the modeler. Thus, firstly we need to detect pageflow gaps. To this end, consider the inference rules depicted in Listings 7.16, 7.17, and 7.18.

```
[detectDanglingPage:
  (?p rdf:type dialog:Page),
  noValue(?t1 usmo:follows ?p),
  noValue(?t2 usmo:precedes ?p) ->
    (?p rdf:type dialog:DanglingPage)]
```

Listing 7.16: *Inference rule to detect dangling pages*

```
[detectRightGap:
  (?p rdf:type dialog:Page),
  noValue(?t1 usmo:follows ?p),
  (?t2 usmo:precedes ?p) ->
    (?p rdf:type dialog:RightGap)]
```

Listing 7.17: *Inference rule to detect pages without following pages*

```
[detectLeftGap:
  (?p rdf:type dialog:Page),
  noValue(?p rdf:type dialog:StartPage),
  (?t1 usmo:follows ?p),
  noValue(?t2 usmo:precedes ?p) ->
    (?p rdf:type dialog:LeftGap)]
```

Listing 7.18: *Inference rule to detect pages without preceding pages (except start page)*

Listing 7.16 shows a Jena inference rule that detects dangling pages and classifies them accordingly. A DanglingPage is a page that is not linked with any Transition. Listing 7.17 and 7.18 show Jena inference rules that detect pages and that are linked only in one direction. Here, we explicitly exclude StartPages, which are of course allowed to be unreferenced because they denote the actual pageflow begin. Having properly classified all candidates for repair operations brings us to the next task, which is the specification of an appropriate adaptation rule that repairs pageflow gaps. The rule depicted in Listing 7.19 shows an example.

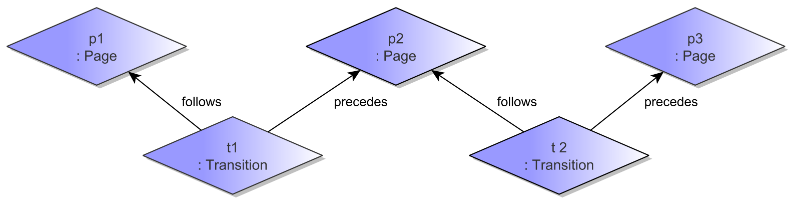
```
[closePageflow:
  (?p2 rdf:type dialog:RightGap),
  (?p1 rdf:type dialog:LeftGap)  ->
    [(?t usmo:precedes ?p2) <-
      makeInstance(?t, usmo:follows, usmo:Transition, ?p1)]]
```

Listing 7.19: *Adaptation rule to detect and repair gaps in the pageflows*

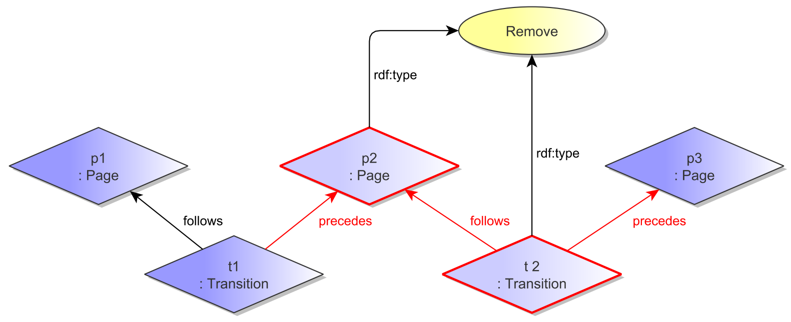
The definition of adaptation rules, and especially of those that shall be executed automatically, must be done with care and foresight. The example in Listing 7.19, for instance, connects arbitrary dangling pages and gaps with each other. Thus, the rendering of repair proposals in the workbench UI would be more appropriate than triggering automatic execution in this case. Another solution is an explicit tracking of the modification actions performed by the feature mapping tool. In case the introduction of a gap between linked pages is tracked, it is very likely that this gap shall be closed. Figure 7.53 illustrates the process. Figure 7.53 (a) shows the initial state with a consistent pageflow, Figure 7.53 (b) shows the knowledgebase after variant creation, where appropriate tooling has classified removed elements as Remove, and Figure 7.53 (c) shows the knowledgebase after having applied a dedicated adaptation rule.

Limitations

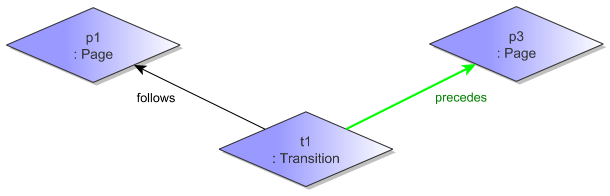
In [HCW07], Hessellund et al. show a guidance examples based on models that are entirely mapped into a Prolog knowledgebase. The examples elucidate the general limits of guidance within HybridMDSD. Since we map only the ontological commitment of domain-specific models into the ontology technological space instead of projecting all contents, also guidance possibilities are restricted to this partial representation.



(a) Knowledgebase before variant creation



(b) Knowledgebase after variant creation with detected changes



(c) Knowledgebase after execution of adaptation rule

Figure 7.53: *Model repair actions with modification detection*

7.3.3 Generating Integration Structures

One of the most outstanding HybridMDSD features is the ability to produce code composition structures based on model reference information. In Section 7.2.3 we comprehensively presented how different language modules are integrated on code level. The definition of composition generators allows to precisely how diverse DSLs reference each other in control and data flows. This even counts for more complex [n:m] relations as shown in Section 7.2.3.5.

The power of code level integration based on a semantic sound foundation paves the way for purely model-driven software development. Depending on the complexity of employed DSLs and code generators it is possible to weave arbitrary application code in various ways. To this end, we showed only *one* possible way to define composition generators based on openArchitectureWare generator templates that produce AOP code. Thus, it is perfectly well possible to employ *any other* template engine as well as composition system in order to produce necessary integration structures. Hence, the power of the HybridMDSD code composition component is promising for arbitrary integration scenarios.

7.4 Discussion

After the presentation of the practical validation we discuss the overall results as well as the experiences that we made during development.

7.4.1 General Feasibility and Appropriateness

The previous sections verified that the vision of the multi-domain engineering paradigm can be realized on the basis of semantic foundations and corresponding reasoning features. We demonstrated the possibility to integrate diverse modeling stacks with an appropriate loose coupling mechanism (see Section 7.1.2.1). Moreover, the abstract treatment of the code composition task allows to compose even most heterogenous platform implementations depending on the composition system.

Nevertheless, it is questionable whether the rather expensive preliminary work is justified. After all there are several development steps necessary before the actual software production can begin. Domain-specific languages need to be created, appropriate code generators must be developed, a semantic mapping has to be provided, and the actual integration of different language modules is to be specified. In addition, extra development roles are involved in this process that are usually not necessary in traditional software development.

In this respect, it must be stated that the expensive preliminary work is *not* necessary for each and every development project. Instead, the HybridMDSD approach allows to systematically establish a flexible modeling infrastructure in software companies. Developed DSLs can be reused in various contexts. Additional code generators can provide unforeseen platform implementations. Corresponding code composition patterns likewise. All these features are accompanied by semantically qualified development assets, which allows for efficient language retrieval capabilities, provides additional documentation, and paves the way for improved integration possibilities with prospective systems.

Altogether results in an increasingly maximized value of once created software development assets and leads to a continuously extending range of services that an IT company can offer.

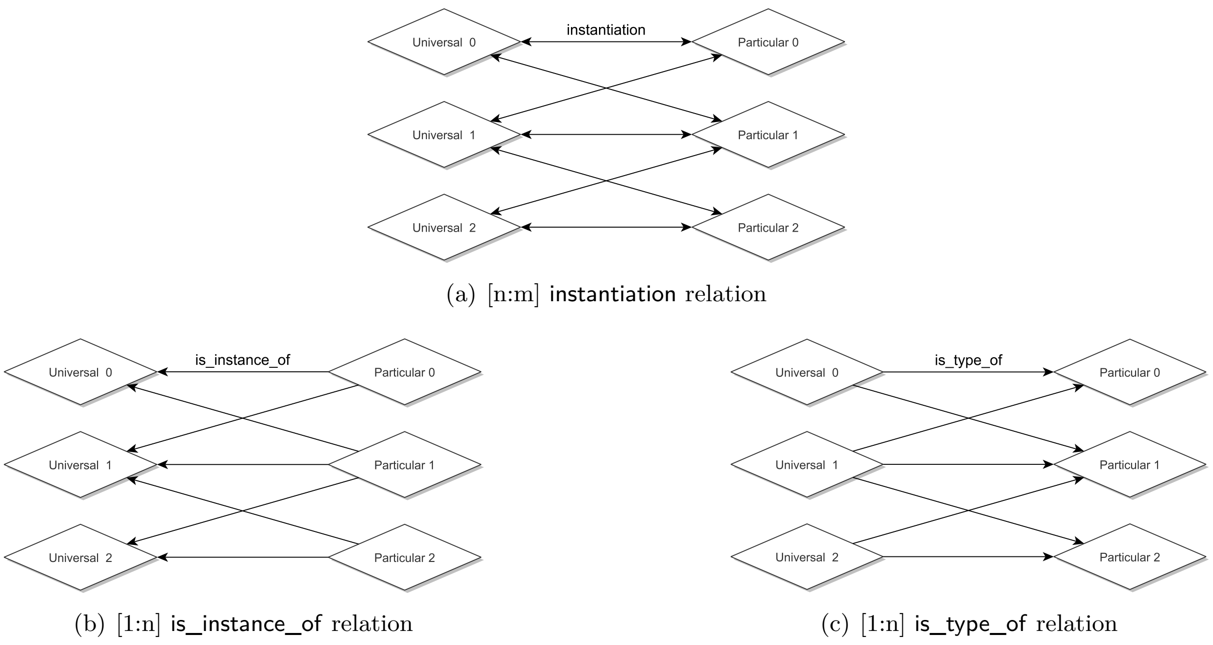
7.4.2 Complex [n:m] Relations

An interesting question to analyze in context of the integration of multiple languages into one single project was whether [n:m] relationships exist and how these relationships are resolved. This is because such relations impose additional complexity to the realizing architecture. The experiences we made in our case study implementations are very interesting. We identified two different types of [n:m] relations within HybridMDSD, namely *ontological* and *code composition* relations, where each refers to the corresponding level of abstraction.

Ontological [n:m] relations

Firstly, ontological [n:m] relations are semantic relations that exist between multiple individuals. An example is the *instantiation* relation illustrated in Figure 7.54 (a). One Particular can be instance of multiple Universals and one Universal can be the intensional definition for several Particulars. Without dedicated consideration, we often indirectly omitted the complexity of such relations by splitting them into relational pairs. Thus, in USMO the [n:m] instantiation relation is respresented with the two properties is_instance_of and is_type_of, as shown in Figure 7.54 (b) and (c).

However, there are other cases where complex relations are established employing a dedicated concept. An example is the USMO concept Relator that links various Substantials as depicted in Figure 7.55. In such cases we actually model [1:n] relations between the *connecting* individual and the *connected* elements, but the actual relation semantically represents an [n:m] relation, namely the *cartesian product*, between all linked individuals.



(a) [n:m] instantiation relation

(b) [1:n] is_instance_of relation

(c) [1:n] is_type_of relation

Figure 7.54: *Ontological [n:m] relations*

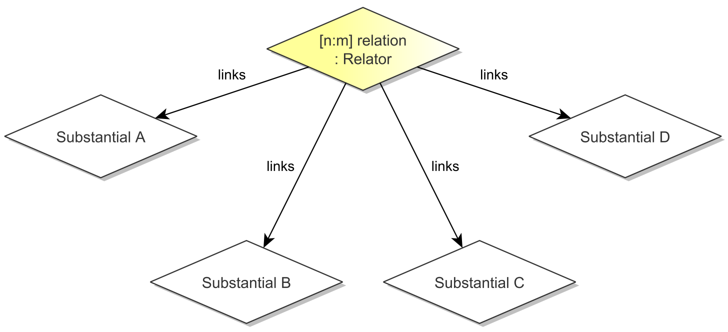


Figure 7.55: *Ontological [n:m] relations with dedicated concepts*

Code Composition [n:m] relations

Secondly, with code composition [n:m] relations we denote actual composition patterns that relate the generated code of multiple language modules to each other. The representation of such relations exists within HybridMDSD composition generators. Here, complex relations need to be resolved to a sequential chain of programming code statements. Figure 7.56 illustrates this context. The figure's background shows five different language modules where each contains a generated fragment box with corresponding hooks. In the figure's foreground, knowledgebase individuals are sketched that form a chain of [1:1] references between individuals. The composition generator that is attached to this reference chain, shown in the figure's center, finally composes all generated fragment boxes and denotes, thus, an [n:m] relation.

As an example recall the integration scenario presented in Section 7.2.3.5. The knowledgebase excerpt shown in Figure 7.48 on page 178 illustrates exclusively 1:1 relations. However, depicted relations result in an integration scenario that involves all generated assets of participating language modules in one single place, namely the composition target. As shown in the composition generator (see Listing 6.2) and the resulting integration code (see Listing 7.14), this is the place where the triggering UI widget event is catched. Here, the invocation of additional business logic takes place and all connected language modules are involved one after another.

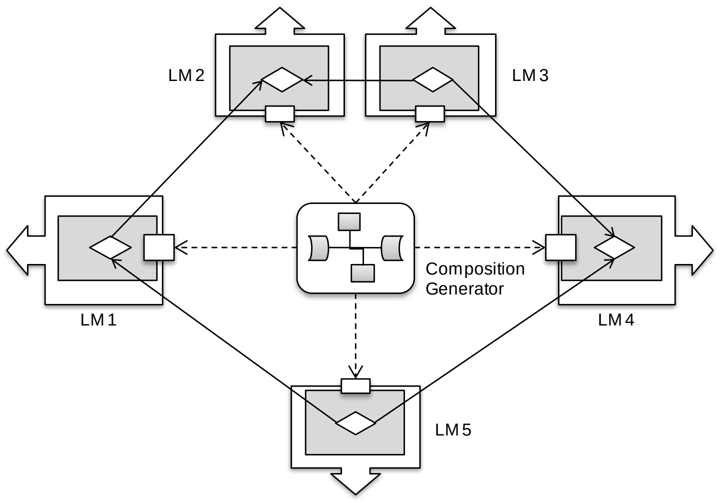


Figure 7.56: *Code composition [n:m] relations*

7.4.3 Implications of Semantic Web Technologies

In context of the prototypical implementation of HybridMDSD we choose Semantic Web technologies such as OWL-DL, the Pellet DL reasoner, and the Jena rule engine as enabling means to realize the semantic foundation. However, especially OWL imposes a number of challenges in this respect which need to be discussed.

Thus, the web ontology language does not use the *unique name assumption (UNA)* [SWM04]. The rather common assumption in informatics has an effect on the identity between individuals. Following the UNA entails that individuals are identified by their name. Not using the assumption implies that two different names may perfectly well refer to the same individual. Moreover, OWL makes an *open world assumption (OWA)* which means that everything that is not explicitly stated must not necessarily be wrong. Instead, unasserted information might exist in some unknown ontology or knowledgebase. This new information may be contradictory but cannot retract existing facts. Hence, the consequences of additional data are *monotonic*.

Since software models in general follow the *closed world assumption (CWA)* and are *prescriptive* specifications rather than *descriptive* information, the implementation of the HybridMDSD core with OWL imposed some challenges, which we have already publicly discussed in [BL08]. In summary, to get a DL reasoner ready to yield sensible information about system integrity that has value for a software developer, employed semantic assets need to be equipped with additional assertions that result in tightly limited knowledgebases. This way, a closed world can be simulated and desired reasoning results can be produced.

To this end it is necessary to equip project knowledgebases with additional facts, such as equality and disparity between individuals or class enumerations which completely define concept extensions by setting them as enumeration of all available individuals. For a detailed presentation of this topic the reader is kindly referred to [BL08].

Beside the extra efforts that result from an open world semantic foundation, the OWA has also advantages with respect to reuse. Since HybridMDSD inter-language connections are based on these semantic structures, the integration potential is maximized because implicit limitations and restrictions are omitted. In contrast, if we would commit, for instance, to the UNA an integration of language modules with equally named language ontology classes would be impossible. Thus, there are pros and cons in using an open world view in our context.

7.4.4 Degree of Centralization

In the previous sections, we repeatedly stated that the employment of a central upper ontology minimizes the efforts that are necessary for language integration. The mapping of modeling constructs to one single semantic core should improve the integration task by providing implicit relationships, which results in advantages compared to 1:1 approaches such as model transformations (cf. Section 5.2.1).

Now, the experiences we made in our case study implementation revealed that this is not completely the case. Finally, dedicated efforts for the integration of languages persist and can not be omitted only because the composed language ontologies inherit from one central upper ontology. Although an implicit relationship between integrated DSLs exist through the ontology, it turned out necessary and valuable to make these relations explicit. Since already the naming of particular inter-language connections was bound to the project scope, we needed to specify dedicated properties that derived from USMO properties. Moreover, in case of complex connections it is necessary to provide extra tooling that realizes the instantiation of bridging individuals, as illustrated in Section 7.2.3.5.

Nevertheless, the necessary integration efforts on semantic level is manageable and does not harm the resulting advantages significantly.

7.4.5 Reasoning Architecture

The introduced architecture of HybridMDSD follows a loose coupling approach to reach a maximum of flexibility with respect to the employed modeling technology stack and facilitate the employment of Semantic Web technologies in software engineering (see Section 4.1.2). Ontology individuals are equipped with reference data that identifies particular constructs in the modeling technological space.

Although this approach is advantageous for the resulting solution, it produces additional challenges in maintenance and projection for the realizing architecture. Thus, sophisticated tooling is necessary to observe the modeling landscape for changes and trigger appropriate mapping handlers that produce a parallel representation of solution space assets in the ontology technological space. Nevertheless, this facilitates the use of comprehensive instance level reasoning power which proved a valueable contribution to common modeling technologies. Together with the strong integration into the Eclipse IDE that provides a rich set of features to work up and present actual reasoning results, the complex architecure pays for itself.

Chapter 8

Summary, Conclusion, and Outlook

8.1 Summary

In this thesis, we presented a novel approach to develop variability-supporting software in a purely model-driven way. To this end, we comprehensively analyzed actual problems in software development and found out that current software projects commonly suffer from both, a high degree of technical complexity and the variety of customer needs. Based on these findings we created a problem and a goal hierarchy following the ZOPP approach and derived our vision of *multi-domain engineering.* We consolidated our preliminary considerations and stated particular requirements that approaches need to fulfill for an implementation of MDE.

The identified requirements facilitated the design of an implementing solution approach — HybridMDSD. We presented the approach in its intentional and technical dimensions, showed its general application with an illustrative example, and enumerated the expected benefits. As a next step, we reviewed published approaches that are related to HybridMDSD. We focussed on work that aims at a comprehensive covering of the entire software engineering process — from higher-level requirements, over solution specifications, to executable applications — presented different approaches to model composition, and introduced projects that employ ontologies in software engineering.

Successively, we presented the results of our research work. We introduced an *ontology for software models* based on a comprehensive analysis of existing upper ontologies in the area of discourse, a classification of domain-specific languages, and the light-weight ABC ontology. In addition, we introduced our *solution to automate the code composition* task that is necessary when integrating generated assets of arbitrary DSLs. Moreover, we described an *extensive HybridMDSD development process* that integrates all solution parts and arranges them from an activity-centric perspective.

Finally, we verified the feasibility of our conceptual approach with a comprehensive prototypical implementation based on the Eclipse IDE. Additionally, we evaluated both tooling and approach with an industrial-scale case study that involved six diverse DSLs, corresponding code generators, ontological mappings, and code composition assets. We demonstrated the features of our approach and discussed our experiences.

8.2 Conclusion

To conclude, we revisit the goal hierarchy introduced in Section 3.3.1. For this purpose, we take a stance on the important goals listed below.

Raised Level of Abstraction

Without a doubt, the employment of model-driven technologies raises the level of abstraction in software development. Here, the HybriMDSD paves the way for an effective application of MDSD ideas. The holistic treatment of assets that are necessary for model-driven development and the demand for self-containedness allows for sustainable MDSD workbenches.

Domain Specialization of in Development

The orientation towards domain specialization is implemented with a proper separation of development concerns into dedicated areas of expertise. Especially the restriction of modeling languages to dedicated problem domains allows for effective application of model-driven technologies. HybridMDSD focuses on domain specialization with a subdivision of complex software solution spaces into particular language modules.

Moderate Change Integration Costs

We are not able to provide empirical data that allows to measure the cost of resources that emerge with new requirements in running projects. To do so, it would be necessary to execute long-term studies that involve multiple projects and stakeholders. However, the sophisticated reasoning facilities of the HybridMDSD semantic foundation proved valuable already in one explicitly complex use case. It enabled us to reveal missing inter-model references and incorrect feature mappings. Moreover, the sophisticated support for automated production of integrating code structures reduced the amount of necessary efforts dramatically.

Support for Individual Requirements

The HybridMDSD approach welcomes changes, as explained before. Our domain integration architecture explicitly deals with modification by maintaining solution space wide integrity constraints and advanced adaptation rules. Thus, our approach is an ideal platform to develop applications that underlie more or less heavy modifications which are, e.g., caused by customer-centric feature choices.

8.3 Outlook

In this last section, we propose several directions for future work that extends the HybridMDSD approach for multi-domain engineering.

Derivation of Composition Generators

Being able to automate the production of assets that compose separate platform code generated from diverse DSLs is already an important milestone in model-driven development. However, correspondig composition generators still need to be specified manually although particular information about the relation of languages is available. The next step towards automation is the *derivation* of composition generators. To this end, the following data can be used as basis for analyses:

Language Relationships: The deduced knowledge available from the semantic foundation gives information about how particular modeling constructs are related in kind and complexity. For further research it would be interesting whether recurring patterns between code composition and language relations exist.

Platform Information: The generator profiles of HybridMDSD language modules provide meta-data about the programming code that is produced. This information is useful to identify what composition technologies and systems are applicable for dedicated composition cases.

Trace Models: One important element that finally facilitates the derivation of composition generators can be provided by trace models. Thus, it is necessary to properly capture how particular model elements are represented in generated assets. A tracing component captures this information and allows to identify lines, sections, and characters in produced assets. Together with the platform information mentioned above, such location data can be interpreted and processed accordingly. This paves the way for the deduction of information about the data and control flow in generated applications.

Integration of Interpreters

In this thesis, we restricted our consideration to the integration of generated programming code. However, there are also use cases where model interpreters are applied to yield executable applications. In this respect, an interesting direction for future research would be the application of presented concepts in context of model interpretation.

HybridMDSD at Runtime

The consideration of interpreters opens the door for another worthwhile research direction. Currently, HybridMDSD addresses exclusively the *design time* in software engineering. In case of utilizing model interpreters, all of the introduced concepts could be applied also during *runtime*. With appropriate tooling that facilitates the invocation of code composition when software is actually up and running, runtime-bindable product features could be realized.

Appendix A

Unified Software Modeling Ontology

A.1 Concepts and Relationships in N3

The following listing shows the implementation of the Unified Software Modeling Ontology. We exported the source from Protégé in *Notation 3 (N3)* [BL06] for the sake of brevity.

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix usmo:    <http://www.sap.com/research/cgt/usmo#> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

usmo:Particular
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Entity ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Particular ;
                owl:onProperty usmo:contains
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Class ;
                owl:complementOf usmo:Universal
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Particular ;
                owl:onProperty usmo:is_part_of
              ] ;
      owl:disjointWith usmo:Universal ;
      owl:equivalentClass
              [ rdf:type owl:Class ;
                owl:unionOf (usmo:Action usmo:Event usmo:Life usmo:Moment usmo:Process usmo:Representative
                     usmo:Situation usmo:Substantial usmo:TemporalQuality usmo:Transition)
              ] .

usmo:depends_on
      rdf:type owl:TransitiveProperty , owl:ObjectProperty ;
      owl:inverseOf usmo:has_dependent .

usmo:triggers
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Event ;
      rdfs:range
              [ rdf:type owl:Class ;
                owl:unionOf (usmo:Action usmo:Transition)
              ] ;
      rdfs:subPropertyOf usmo:has_dependent ;
      owl:inverseOf usmo:is_triggered_by .

usmo:may_trigger
      rdf:type owl:TransitiveProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Event ;
      rdfs:range usmo:Behavior ;
      rdfs:subPropertyOf usmo:has_dependent ;
      owl:inverseOf usmo:may_be_triggered_by .

usmo:links
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Relator ;
      rdfs:range usmo:Substantial ;
      rdfs:subPropertyOf usmo:depends_on ;
      owl:inverseOf usmo:is_linked_by .

usmo:is_involved_in
      rdf:type owl:ObjectProperty ;
      rdfs:domain
              [ rdf:type owl:Class ;
                owl:unionOf (usmo:Moment usmo:Substantial)
              ] ;
      rdfs:range usmo:Action ;
      rdfs:subPropertyOf usmo:has_dependent ;
      owl:inverseOf usmo:involves .

usmo:has_event
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Situation ;
      rdfs:range usmo:Event ;
      rdfs:subPropertyOf usmo:has_dependent ;
      owl:inverseOf usmo:occurs_in .

usmo:Universal
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Entity ;
      rdfs:subClassOf
              [ rdf:type owl:Class ;
                owl:complementOf usmo:Particular
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Universal ;
                owl:onProperty usmo:contains
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Universal ;
                owl:onProperty usmo:depends_on
              ] ;
      owl:disjointWith usmo:Particular ;
      owl:equivalentClass
              [ rdf:type owl:Class ;
                owl:unionOf (usmo:Behavior usmo:Property usmo:Schema usmo:Signal usmo:TemporalProperty)
              ] .

usmo:Actuality
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Entity ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Actuality ;
                owl:onProperty usmo:is_part_of
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Actuality ;
                owl:onProperty usmo:contains
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Class ;
                owl:complementOf usmo:Temporality
              ] ;
      owl:disjointWith usmo:Temporality ;
      owl:equivalentClass
              [ rdf:type owl:Class ;
                owl:unionOf (usmo:Moment usmo:Property usmo:Representative usmo:Schema usmo:Substantial)
              ] .

usmo:represents
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Representative ;
      rdfs:range
              [ rdf:type owl:Class ;
                owl:unionOf (usmo:Particular usmo:Set)
              ] ;
      rdfs:subPropertyOf usmo:depends_on ;
      owl:inverseOf usmo:is_represented_by .

usmo:Activity
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Situation ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Activity ;
                owl:onProperty usmo:contains
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Activity ;
                owl:onProperty usmo:is_part_of
              ] ;
      owl:disjointWith usmo:State .

usmo:RelationalProperty
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Property ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:minCardinality "2"^^xsd:int ;
                owl:onProperty usmo:relates
              ] ;
      rdfs:subClassOf
```

```
                [ rdf:type owl:Restriction ;
                  owl:allValuesFrom usmo:Relator ;
                  owl:onProperty usmo:has_instance
                ] ;
        owl:disjointWith usmo:IntrinsicProperty .

usmo:Property
        rdf:type owl:Class ;
        rdfs:subClassOf usmo:Actuality , usmo:Universal ;
        rdfs:subClassOf
                [ rdf:type owl:Restriction ;
                  owl:onProperty usmo:depends_on ;
                  owl:someValuesFrom usmo:Schema
                ] ;
        rdfs:subClassOf
                [ rdf:type owl:Restriction ;
                  owl:allValuesFrom usmo:Moment ;
                  owl:onProperty usmo:has_instance
                ] ;
        owl:disjointWith usmo:Moment , usmo:Behavior , usmo:Substantial , usmo:Signal , usmo:Schema , usmo:
             Representative ;
        owl:equivalentClass
                [ rdf:type owl:Class ;
                  owl:unionOf (usmo:IntrinsicProperty usmo:RelationalProperty)
                ] .

usmo:has_dependent
        rdf:type owl:TransitiveProperty , owl:ObjectProperty ;
        owl:inverseOf usmo:depends_on .

usmo:Event
        rdf:type owl:Class ;
        rdfs:subClassOf usmo:Particular , usmo:Temporality ;
        rdfs:subClassOf
                [ rdf:type owl:Restriction ;
                  owl:allValuesFrom usmo:Event ;
                  owl:onProperty usmo:contains
                ] ;
        rdfs:subClassOf
                [ rdf:type owl:Restriction ;
                  owl:minCardinality "1"^^xsd:int ;
                  owl:onProperty usmo:triggers
                ] ;
        rdfs:subClassOf
                [ rdf:type owl:Restriction ;
                  owl:allValuesFrom usmo:Signal ;
                  owl:onProperty usmo:is_instance_of
                ] ;
        owl:disjointWith usmo:Behavior , usmo:Action , usmo:Substantial , usmo:Signal , usmo:TemporalProperty ,
             usmo:TemporalQuality , usmo:Transition , usmo:Representative , usmo:Situation , usmo:Life .

usmo:may_be_involved_in
        rdf:type owl:ObjectProperty ;
        rdfs:domain
                [ rdf:type owl:Class ;
                  owl:unionOf (usmo:Property usmo:Schema)
                ] ;
        rdfs:range usmo:Behavior ;
        rdfs:subPropertyOf usmo:has_dependent ;
        owl:inverseOf usmo:may_involve .

usmo:State
        rdf:type owl:Class ;
        rdfs:subClassOf usmo:Situation ;
        rdfs:subClassOf
                [ rdf:type owl:Restriction ;
                  owl:allValuesFrom usmo:State ;
                  owl:onProperty usmo:contains
                ] ;
        rdfs:subClassOf
                [ rdf:type owl:Restriction ;
                  owl:allValuesFrom
                          [ rdf:type owl:Class ;
                            owl:unionOf (usmo:State usmo:Life)
                          ] ;
                  owl:onProperty usmo:is_part_of
                ] ;
        rdfs:subClassOf
                [ rdf:type owl:Restriction ;
                  owl:onProperty usmo:is_part_of ;
                  owl:someValuesFrom usmo:Life
                ] ;
        owl:disjointWith usmo:Activity .

usmo:has_instance
        rdf:type owl:ObjectProperty ;
        rdfs:domain usmo:Universal ;
        rdfs:range usmo:Particular ;
        rdfs:subPropertyOf usmo:has_dependent ;
        owl:inverseOf usmo:is_instance_of .

usmo:Set
        rdf:type owl:Class ;
        rdfs:subClassOf owl:Thing ;
        rdfs:subClassOf
                [ rdf:type owl:Restriction ;
                  owl:minCardinality "1"^^xsd:int ;
                  owl:onProperty usmo:has_member
                ] ;
```

```
      owl:disjointWith usmo:Entity .

usmo:has_phase
      rdf:type owl:TransitiveProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Substantial ;
      rdfs:range usmo:Substantial ;
      rdfs:subPropertyOf usmo:has_dependent ;
      owl:inverseOf usmo:is_phase_of .

usmo:subsets
      rdf:type owl:TransitiveProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Set ;
      rdfs:range usmo:Set ;
      rdfs:subPropertyOf usmo:depends_on ;
      owl:inverseOf usmo:has_subset .

usmo:has_subtype
      rdf:type owl:TransitiveProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Schema ;
      rdfs:range usmo:Schema ;
      rdfs:subPropertyOf usmo:has_dependent ;
      owl:inverseOf usmo:is_subtype_of .

usmo:Signal
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Temporality , usmo:Universal ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Event ;
                owl:onProperty usmo:has_instance
              ] ;
      owl:disjointWith usmo:Action , usmo:Behavior , usmo:Schema , usmo:TemporalProperty , usmo:
           TemporalQuality , usmo:Property , usmo:Transition , usmo:Situation , usmo:Life , usmo:Event .

usmo:has_subset
      rdf:type owl:TransitiveProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Set ;
      rdfs:range usmo:Set ;
      rdfs:subPropertyOf usmo:has_dependent ;
      owl:inverseOf usmo:subsets .

usmo:inheres_in
      rdf:type owl:FunctionalProperty , owl:ObjectProperty ;
      rdfs:domain
              [ rdf:type owl:Class ;
                owl:unionOf (usmo:Quality usmo:TemporalQuality)
              ] ;
      rdfs:range usmo:Particular ;
      rdfs:subPropertyOf usmo:is_part_of ;
      owl:inverseOf usmo:bears .

usmo:relates
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:RelationalProperty ;
      rdfs:range usmo:Schema ;
      rdfs:subPropertyOf usmo:depends_on ;
      owl:inverseOf usmo:is_related_by .

usmo:has_extension
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Universal ;
      rdfs:range usmo:Set ;
      rdfs:subPropertyOf usmo:has_dependent ;
      owl:inverseOf usmo:is_extension_of .

usmo:may_be_triggered_by
      rdf:type owl:TransitiveProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Behavior ;
      rdfs:range usmo:Event ;
      rdfs:subPropertyOf usmo:depends_on ;
      owl:inverseOf usmo:may_trigger .

usmo:has_context
      rdf:type owl:ObjectProperty ;
      rdfs:domain
              [ rdf:type owl:Class ;
                owl:unionOf (usmo:Moment usmo:Representative usmo:Substantial)
              ] ;
      rdfs:range usmo:Situation ;
      rdfs:subPropertyOf usmo:has_dependent ;
      owl:inverseOf usmo:is_context_of .

usmo:Process
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Activity ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Process ;
                owl:onProperty usmo:is_part_of
              ] .

usmo:is_preceded_by
      rdf:type owl:InverseFunctionalProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Situation ;
      rdfs:range usmo:Transition ;
      rdfs:subPropertyOf usmo:has_dependent ;
      owl:inverseOf usmo:precedes .

usmo:is_extension_of
```

```
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Set ;
      rdfs:range usmo:Universal ;
      rdfs:subPropertyOf usmo:depends_on ;
      owl:inverseOf usmo:has_extension .

<http://www.sap.com/research/cgt/usmo>
      rdf:type owl:Ontology ;
      owl:versionInfo "1.0"^^xsd:string .

usmo:is_lived_by
      rdf:type owl:InverseFunctionalProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Life ;
      rdfs:range usmo:Substantial ;
      rdfs:subPropertyOf usmo:depends_on ;
      owl:inverseOf usmo:lives .

usmo:Situation
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Particular , usmo:Temporality ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Situation ;
                owl:onProperty usmo:contains
              ] ;
      owl:disjointWith usmo:Action , usmo:Behavior , usmo:Signal , usmo:TemporalProperty , usmo:
           TemporalQuality , usmo:Transition , usmo:Life , usmo:Event ;
      owl:equivalentClass
              [ rdf:type owl:Class ;
                owl:unionOf (usmo:Activity usmo:State)
              ] .

usmo:is_triggered_by
      rdf:type owl:ObjectProperty ;
      rdfs:domain
              [ rdf:type owl:Class ;
                owl:unionOf (usmo:Action usmo:Transition)
              ] ;
      rdfs:range usmo:Event ;
      rdfs:subPropertyOf usmo:depends_on ;
      owl:inverseOf usmo:triggers .

usmo:may_destroy
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Behavior ;
      rdfs:range usmo:Schema ;
      rdfs:subPropertyOf usmo:may_involve ;
      owl:inverseOf usmo:may_be_destroyed_by .

usmo:is_in
      rdf:type owl:FunctionalProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Substantial ;
      rdfs:range usmo:State ;
      rdfs:subPropertyOf usmo:has_context ;
      owl:inverseOf usmo:is_state_of .

usmo:Representative
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Particular , usmo:Actuality ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Activity ;
                owl:onProperty usmo:has_context
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Representative ;
                owl:onProperty usmo:is_part_of
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Representative ;
                owl:onProperty usmo:contains
              ] ;
      owl:disjointWith usmo:Moment , usmo:Action , usmo:Substantial , usmo:Schema , usmo:TemporalQuality ,
           usmo:Transition , usmo:Property , usmo:Event , usmo:Life .

usmo:is_linked_by
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Substantial ;
      rdfs:range usmo:Relator ;
      rdfs:subPropertyOf usmo:has_dependent ;
      owl:inverseOf usmo:links .

usmo:IntrinsicProperty
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Property ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Quality ;
                owl:onProperty usmo:has_instance
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:onProperty usmo:belongs_to ;
                owl:someValuesFrom
                        [ rdf:type owl:Class ;
                          owl:unionOf (usmo:Schema usmo:Property)
                        ]
              ] ;
      owl:disjointWith usmo:RelationalProperty .

usmo:destroys
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Action ;
      rdfs:range usmo:Substantial ;
      rdfs:subPropertyOf usmo:involves ;
      owl:inverseOf usmo:is_destroyed_by .

usmo:occurs_in
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Event ;
      rdfs:range usmo:Situation ;
      rdfs:subPropertyOf usmo:depends_on ;
      owl:inverseOf usmo:has_event .

usmo:precedes
      rdf:type owl:FunctionalProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Transition ;
      rdfs:range usmo:Situation ;
      rdfs:subPropertyOf usmo:depends_on ;
      owl:inverseOf usmo:is_preceded_by .

usmo:is_member_of
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Particular ;
      rdfs:range usmo:Set ;
      rdfs:subPropertyOf usmo:depends_on ;
      owl:inverseOf usmo:has_member .

usmo:is_instance_of
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Particular ;
      rdfs:range usmo:Universal ;
      rdfs:subPropertyOf usmo:depends_on ;
      owl:inverseOf usmo:has_instance .

usmo:contains
      rdf:type owl:TransitiveProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Entity ;
      rdfs:range usmo:Entity ;
      rdfs:subPropertyOf usmo:has_dependent ;
      owl:inverseOf usmo:is_part_of .

usmo:bears
      rdf:type owl:InverseFunctionalProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Particular ;
      rdfs:range
              [ rdf:type owl:Class ;
                owl:unionOf (usmo:TemporalQuality usmo:Quality)
              ] ;
      rdfs:subPropertyOf usmo:contains ;
      owl:inverseOf usmo:inheres_in .

usmo:is_represented_by
      rdf:type owl:ObjectProperty ;
      rdfs:domain
              [ rdf:type owl:Class ;
                owl:unionOf (usmo:Particular usmo:Set)
              ] ;
      rdfs:range usmo:Representative ;
      rdfs:subPropertyOf usmo:has_dependent ;
      owl:inverseOf usmo:represents .

usmo:Transition
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Particular , usmo:Temporality ;
      owl:disjointWith usmo:Action , usmo:Behavior , usmo:Substantial , usmo:Signal , usmo:TemporalProperty ,
            usmo:TemporalQuality , usmo:Representative , usmo:Situation , usmo:Event , usmo:Life .

usmo:Temporality
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Entity ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Temporality ;
                owl:onProperty usmo:contains
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Class ;
                owl:complementOf usmo:Actuality
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Temporality ;
                owl:onProperty usmo:is_part_of
              ] ;
      owl:disjointWith usmo:Actuality ;
      owl:equivalentClass
              [ rdf:type owl:Class ;
                owl:unionOf (usmo:Action usmo:Behavior usmo:Event usmo:Life usmo:Process usmo:Signal usmo:
                      Situation usmo:TemporalProperty usmo:TemporalQuality usmo:Transition)
              ] .

usmo:may_be_represented_by
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Universal ;
      rdfs:range usmo:Representative ;
```

```
      rdfs:subPropertyOf usmo:has_dependent ;
      owl:inverseOf usmo:may_represent .

usmo:Schema
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Actuality , usmo:Universal ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Substantial ;
                owl:onProperty usmo:has_instance
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:IntrinsicProperty ;
                owl:onProperty usmo:consists_of
              ] ;
      owl:disjointWith usmo:Moment , usmo:Behavior , usmo:Substantial , usmo:Signal , usmo:TemporalProperty ,
           usmo:Property , usmo:Representative .

usmo:lives
      rdf:type owl:FunctionalProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Substantial ;
      rdfs:range usmo:Life ;
      rdfs:subPropertyOf usmo:has_dependent ;
      owl:inverseOf usmo:is_lived_by .

usmo:consists_of
      rdf:type owl:InverseFunctionalProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Universal ;
      rdfs:range
              [ rdf:type owl:Class ;
                owl:unionOf (usmo:TemporalProperty usmo:IntrinsicProperty)
              ] ;
      rdfs:subPropertyOf usmo:contains ;
      owl:inverseOf usmo:belongs_to .

usmo:is_subtype_of
      rdf:type owl:TransitiveProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Schema ;
      rdfs:range usmo:Schema ;
      rdfs:subPropertyOf usmo:depends_on ;
      owl:inverseOf usmo:has_subtype .

usmo:is_followed_by
      rdf:type owl:InverseFunctionalProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Situation ;
      rdfs:range usmo:Transition ;
      rdfs:subPropertyOf usmo:has_dependent ;
      owl:inverseOf usmo:follows .

usmo:is_context_of
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Situation ;
      rdfs:range
              [ rdf:type owl:Class ;
                owl:unionOf (usmo:Moment usmo:Representative usmo:Substantial)
              ] ;
      rdfs:subPropertyOf usmo:depends_on ;
      owl:inverseOf usmo:has_context .

usmo:TemporalQuality
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Temporality , usmo:Particular ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:TemporalQuality ;
                owl:onProperty usmo:contains
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:onProperty usmo:inheres_in ;
                owl:someValuesFrom usmo:Temporality
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:TemporalProperty ;
                owl:onProperty usmo:is_instance_of
              ] ;
      owl:disjointWith usmo:Behavior , usmo:Action , usmo:Substantial , usmo:Signal , usmo:TemporalProperty ,
           usmo:Transition , usmo:Representative , usmo:Situation , usmo:Life , usmo:Event .

usmo:may_create
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Behavior ;
      rdfs:range usmo:Schema ;
      rdfs:subPropertyOf usmo:may_involve ;
      owl:inverseOf usmo:may_be_created_by .

usmo:is_destroyed_by
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Substantial ;
      rdfs:range usmo:Action ;
      rdfs:subPropertyOf usmo:is_involved_in ;
      owl:inverseOf usmo:destroys .

usmo:belongs_to
      rdf:type owl:FunctionalProperty , owl:ObjectProperty ;
      rdfs:domain
              [ rdf:type owl:Class ;
```

```
                owl:unionOf (usmo:IntrinsicProperty usmo:TemporalProperty)
              ] ;
      rdfs:range usmo:Universal ;
      rdfs:subPropertyOf usmo:is_part_of ;
      owl:inverseOf usmo:consists_of .

usmo:TemporalProperty
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Temporality , usmo:Universal ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:TemporalProperty ;
                owl:onProperty usmo:contains
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:onProperty usmo:belongs_to ;
                owl:someValuesFrom usmo:Temporality
              ] ;
      owl:disjointWith usmo:Behavior , usmo:Action , usmo:Signal , usmo:Schema , usmo:TemporalQuality , usmo:
           Transition , usmo:Situation , usmo:Event , usmo:Life .

usmo:may_live
      rdf:type owl:FunctionalProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Schema ;
      rdfs:range usmo:Life ;
      rdfs:subPropertyOf usmo:has_dependent ;
      owl:inverseOf usmo:may_be_lived_by .

usmo:Substantial
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Particular , usmo:Actuality ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:minCardinality "1"^^xsd:int ;
                owl:onProperty usmo:is_instance_of
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Quality ;
                owl:onProperty usmo:bears
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Schema ;
                owl:onProperty usmo:is_instance_of
              ] ;
      owl:disjointWith usmo:Moment , usmo:Action , usmo:Schema , usmo:TemporalQuality , usmo:Property , usmo:
           Transition , usmo:Representative , usmo:Event , usmo:Life .

usmo:Life
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Temporality , usmo:Particular ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom
                        [ rdf:type owl:Class ;
                          owl:unionOf (usmo:State usmo:Transition)
                        ] ;
                owl:onProperty usmo:contains
              ] ;
      owl:disjointWith usmo:Behavior , usmo:Action , usmo:Substantial , usmo:Signal , usmo:TemporalProperty ,
           usmo:TemporalQuality , usmo:Transition , usmo:Representative , usmo:Situation , usmo:Event .

usmo:has_member
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Set ;
      rdfs:range usmo:Particular ;
      rdfs:subPropertyOf usmo:has_dependent ;
      owl:inverseOf usmo:is_member_of .

usmo:may_involve
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Behavior ;
      rdfs:range
              [ rdf:type owl:Class ;
                owl:unionOf (usmo:Property usmo:Schema)
              ] ;
      rdfs:subPropertyOf usmo:depends_on ;
      owl:inverseOf usmo:may_be_involved_in .

usmo:is_phase_of
      rdf:type owl:TransitiveProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Substantial ;
      rdfs:range usmo:Substantial ;
      rdfs:subPropertyOf usmo:depends_on ;
      owl:inverseOf usmo:has_phase .

usmo:is_related_by
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Schema ;
      rdfs:range usmo:RelationalProperty ;
      rdfs:subPropertyOf usmo:has_dependent ;
      owl:inverseOf usmo:relates .

usmo:is_state_of
      rdf:type owl:InverseFunctionalProperty , owl:ObjectProperty ;
      rdfs:domain usmo:State ;
      rdfs:range usmo:Substantial ;
      rdfs:subPropertyOf usmo:is_context_of ;
      owl:inverseOf usmo:is_in .

usmo:Behavior
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Temporality , usmo:Universal ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Action ;
                owl:onProperty usmo:has_instance
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:TemporalProperty ;
                owl:onProperty usmo:consists_of
              ] ;
      owl:disjointWith usmo:Action , usmo:Signal , usmo:Schema , usmo:TemporalProperty , usmo:TemporalQuality
           , usmo:Transition , usmo:Property , usmo:Situation , usmo:Event , usmo:Life .

usmo:is_created_by
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Substantial ;
      rdfs:range usmo:Action ;
      rdfs:subPropertyOf usmo:is_involved_in ;
      owl:inverseOf usmo:creates .

usmo:is_part_of
      rdf:type owl:TransitiveProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Entity ;
      rdfs:range usmo:Entity ;
      rdfs:subPropertyOf usmo:depends_on ;
      owl:inverseOf usmo:contains .

usmo:Relator
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Moment ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:minCardinality "2"^^xsd:int ;
                owl:onProperty usmo:links
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:RelationalProperty ;
                owl:onProperty usmo:is_instance_of
              ] ;
      owl:disjointWith usmo:Quality .

usmo:may_be_destroyed_by
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Schema ;
      rdfs:range usmo:Behavior ;
      rdfs:subPropertyOf usmo:may_be_involved_in ;
      owl:inverseOf usmo:may_destroy .

usmo:Action
      rdf:type owl:Class ;
      rdfs:subClassOf usmo:Particular , usmo:Temporality ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:TemporalQuality ;
                owl:onProperty usmo:bears
              ] ;
      rdfs:subClassOf
              [ rdf:type owl:Restriction ;
                owl:allValuesFrom usmo:Behavior ;
                owl:onProperty usmo:is_instance_of
              ] ;
      owl:disjointWith usmo:Behavior , usmo:Substantial , usmo:Signal , usmo:TemporalProperty , usmo:
           TemporalQuality , usmo:Transition , usmo:Representative , usmo:Situation , usmo:Event , usmo:Life .

usmo:Entity
      rdf:type owl:Class ;
      owl:disjointWith usmo:Set ;
      owl:equivalentClass
              [ rdf:type owl:Class ;
                owl:intersectionOf ([ rdf:type owl:Class ;
                            owl:unionOf (usmo:Actuality usmo:Temporality)
                          ] [ rdf:type owl:Class ;
                            owl:unionOf (usmo:Particular usmo:Universal)
                          ])
              ] .

usmo:creates
      rdf:type owl:ObjectProperty ;
      rdfs:domain usmo:Action ;
      rdfs:range usmo:Substantial ;
      rdfs:subPropertyOf usmo:involves ;
      owl:inverseOf usmo:is_created_by .

usmo:follows
      rdf:type owl:FunctionalProperty , owl:ObjectProperty ;
      rdfs:domain usmo:Transition ;
      rdfs:range usmo:Situation ;
      rdfs:subPropertyOf usmo:depends_on ;
      owl:inverseOf usmo:is_followed_by .

usmo:may_be_created_by
      rdf:type owl:ObjectProperty ;
```

```
        rdfs:domain usmo:Schema ;
        rdfs:range usmo:Behavior ;
        rdfs:subPropertyOf usmo:may_be_involved_in ;
        owl:inverseOf usmo:may_create .

usmo:Moment
        rdf:type owl:Class ;
        rdfs:subClassOf usmo:Particular , usmo:Actuality ;
        rdfs:subClassOf
                [ rdf:type owl:Restriction ;
                  owl:minCardinality "1"^^xsd:int ;
                  owl:onProperty usmo:is_instance_of
                ] ;
        rdfs:subClassOf
                [ rdf:type owl:Restriction ;
                  owl:allValuesFrom usmo:Property ;
                  owl:onProperty usmo:is_instance_of
                ] ;
        rdfs:subClassOf
                [ rdf:type owl:Restriction ;
                  owl:onProperty usmo:depends_on ;
                  owl:someValuesFrom usmo:Substantial
                ] ;
        owl:disjointWith usmo:Substantial , usmo:Schema , usmo:Property , usmo:Representative ;
        owl:equivalentClass
                [ rdf:type owl:Class ;
                  owl:unionOf (usmo:Quality usmo:Relator)
                ] .

usmo:involves
        rdf:type owl:ObjectProperty ;
        rdfs:domain usmo:Action ;
        rdfs:range
                [ rdf:type owl:Class ;
                  owl:unionOf (usmo:Moment usmo:Substantial)
                ] ;
        rdfs:subPropertyOf usmo:depends_on ;
        owl:inverseOf usmo:is_involved_in .

usmo:may_be_lived_by
        rdf:type owl:InverseFunctionalProperty , owl:ObjectProperty ;
        rdfs:domain usmo:Life ;
        rdfs:range usmo:Schema ;
        rdfs:subPropertyOf usmo:depends_on ;
        owl:inverseOf usmo:may_live .
```

Listing A.1: *The Unified Software Modeling Ontology in Notation 3*

A.2 Axiomatization with Jena Rules

The listing below shows a selection of the axiomatization of USMO, which we firstly introduced in [BL07]. This comprises integrity rules for structure and behavior as well as additional deductive rules.

```
# Axioms for the Unified Software Modeling Ontology (USMO)

@prefix usmo: <http://www.sap.com/research/cgt/usmo#>.
@prefix sc: <http://www.sap.com/research/cgt/sc#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix dialog: <http://dialog.salesscenario.feasiple.de#>.

# we do not include the OWL rules if we use Pellet for OWL reasoning
# @include <OWL>.


#-------------------#
# Integrity rules.  #
#-------------------#

# -- structural integrity
#

# The dependency relation is anti-symmetric and irreflexive to avoid dependency
# cycles.
[validateCircularDependency:
   (?e1 usmo:depends_on ?e2),(?e2 usmo:depends_on ?e1) ->
      (?e1 rdf:type sc:Invalid),
      (?e1 sc:violates 'Circular dependency'),
      (?e1 sc:caused_by ?e2)]

# An Entity cannot be contained by two entities unless these are related via
# containment themselves (transitive containment)
[validateTransitiveContainment:
   (?e1 usmo:contains ?e3), (?e2 usmo:contains ?e3), (?e1 owl:differentFrom ?e2),
        noValue(?e1 usmo:contains ?e2), noValue(?e2 usmo:contains ?e1) ->
      (?e3 rdf:type sc:Invalid),
      (?e3 sc:violates 'Entity contained by two unrelated entities'),
      (?e3 sc:caused_by ?e1),(?e3 sc:caused_by ?e1) ]

# An Entity e3 that belongs to an Entity e1 cannot be contained by a third
# Entity e2 unless e2 also contains e1 (i.e., composition must bind stronger
# than containment)
[validateComposition:
   (?e1 usmo:consists_of ?e3), (?e2 usmo:contains ?e3), notEqual(?e1,?e2),
        noValue(?e2 usmo:contains ?e1) ->
      (?e2 rdf:type sc:Invalid),
      (?e2 sc:violates 'Entity contains another entity E, but does not contain the entity which E belongs to,
           thereby breaking its composition relationship'),
      (?e2 sc:caused_by ?e3),(?e2 sc:caused_by ?e1)]

# This is a test integrity rule to check the new 'notExists' built-in which
# allows to do a non-monotonous query over the database simulating closed
# world semantics
[validateDependency:
  (?p rdf:type usmo:Property),
        notExists(?u rdf:type usmo:Universal, ?p usmo:depends_on ?u) ->
      (?p rdf:type sc:Invalid),
      (?p sc:violates 'Property does not depend on a universal')]


# If a Universal u has an extension ext, then there must be at least one Particular p in ext.
[validateNonEmptyExtensions:
   (?u rdf:type usmo:Universal),
        notExists(?u rdf:type usmo:Universal, ?p usmo:depends_on ?u) ->
      (?p rdf:type sc:Invalid),
      (?p sc:violates 'Property does not depend on a universal')]

# For each Quality q that inheres in a Substantial (or another Moment) x with
# x being an instance of a Schema (or Property) y, there must be an Intrinsic-
# Property i such that q instantiates i and i belongs to y. Alternatively, if
# y has a supertype z, then i can also belong to z, because properties are
# inherited by subtypes:
[validatePropertyExistsForQuality:
  (?q rdf:type usmo:Quality), (?q usmo:inheres_in ?x), (?x usmo:is_instance_of ?y),
      notExists(?y2 usmo:is_subtype_of ?y, ?x usmo:is_instance_of ?y2),
      notExists(?i rdf:type usmo:IntrinsicProperty, ?q usmo:is_instance_of ?i,
        ?i usmo:belongs_to ?y),
      notExists(?i rdf:type usmo:IntrinsicProperty, ?q usmo:is_instance_of ?i,
        ?z usmo:has_subtype ?y, ?i usmo:belongs_to ?z) ->
    (?q rdf:type sc:Invalid),
    (?q sc:violates 'There is no matching intrinsic property for this quality'),
    (?q sc:caused_by ?x)]

# A Representative must either represent at least one Particular, or alternatively modally represent
# at least one Universal. In other words, purely decorative elements should not be mapped into
# the Ontology TS.
[validateNonDanglingRepresentatives:
  (?r rdf:type usmo:Representative),
        notExists(?r usmo:represents ?p, ?r usmo:may_represent ?u) ->
      (?p rdf:type sc:Invalid),
      (?p sc:violates 'Representatives does not represent anything')]

# -- behavioral integrity
#

# If an Action a is instance of some Behavior b and a involves a Substantial s than there must be
# some Schema that is involved by b and that is type of s:
[validateBehaviourTypeSafety:
   (?a usmo:involves ?s),
   (?a usmo:is_instance_of ?b),
   (?s usmo:is_instance_of ?sc1),
   noValue(?b usmo:may_involve ?sc1) ->
      (?a rdf:type sc:Invalid),
      (?a sc:violates 'Action involves instance of wrong or not defined type'),
      (?a sc:caused_by ?s)]

# A Particular p cannot be in the context of more than one Situation, unless the Situations form a
# containment relationship.
[validateParticularInOneSituation:
   (?p usmo:has_context ?s1),
   (?p usmo:has_context ?s2),
   noValue(?s1 owl:sameAs ?s2) ,
   noValue(?s1 usmo:contains ?s2) ->
      (?p rdf:type sc:Invalid),
      (?p sc:violates 'Particulars cannot be in more than one Situations at a time'),
      (?p sc:caused_by ?s1)]

# If a Substantial s lives a Life l and s is instance of a Schema sc, then sc needs to modally live
# l. Likewise, if a Schema sc may live a Life l and is instantiated by a set of Substantials, then
# there must exist at least one Substantial s that actually lives l (i.e., is the universal facet of all
# ontically connected Substantials).
[validateLifeDefinition:
   (?s usmo:lives ?l),
   (?s usmo:is_instance_of ?sc),
   noValue(?sc usmo:may_live ?l) ->
      (?s rdf:type sc:Invalid),
      (?s sc:violates 'Substantial lives Life that its Schema is not defined to life'),
      (?s sc:caused_by ?l)]

[validateLifeDefinition:
   (?sc usmo:may_live ?l),
   (?sc usmo:has_extension ?ext),
   (?s usmo:is_instance_of ?sc),
   noValue(?s usmo:lives ?l) ->
      (?sc rdf:type sc:Invalid),
      (?sc sc:violates 'Substantial does not live Life as defined by its Schema'),
      (?sc sc:caused_by ?l)]

# A Substantial cannot be phase of several other Substantials, unless these are also related via a
# facet separation relationship:
[validateSubstantialPhases:
   (?s1 usmo:is_phase_of ?s2),
   (?s1 usmo:is_phase_of ?s3),
   noValue(?s2 owl:sameAs ?s3),
   noValue(?s2 usmo:is_phase_of s3) ->
      (?s1 rdf:type sc:Invalid),
      (?s1 sc:violates 'Substantial can phase of only one other Substantial'),
      (?s1 sc:caused_by ?s3)]

# A Substantial that is a phase of another Substantial must not live a Life, because it is only an
# existential facet. In addition, Substantials that represent existential facets must always be in a
# State. Likewise, if a Substantial is in a State, it must be a phase of some other Substantial (either
# another existential facet or the universal facet). Similarly, a Substantial that lives a Life must
# not be a phase of another Substantial, because it is the universal facet. Finally, Substantials that
# represent universal facets must never be in a State:
[validateExistentialFacetLivesLife:
   (?s1 usmo:is_phase_of ?s2),
   noValue(?s1 usmo:lives ?l) ->
      (?s1 rdf:type sc:Invalid),
      (?s1 sc:violates 'Existential facet of Substantial does not live a Life')]

[validateExistentialFacetIsInState:
   (?s1 usmo:is_phase_of ?s2),
   (?st rdf:type usmo:State)
   noValue(?s1 usmo:is_in ?st) ->
      (?s1 rdf:type sc:Invalid),
      (?s1 sc:violates 'Existential facet of Substantial is not in a State')]

[validateExistentialFacetDangling:
   (?s1 usmo:is_in ?st),
   noValue(?s1 usmo:is_phase_of ?s2) ->
      (?s1 rdf:type sc:Invalid),
      (?s1 sc:violates 'Existential facet of Substantial is dangling')]

[validateUniversalFacet:
   (?s usmo:lives ?l),
   noValue(?s1 usmo:is_in ?st) ->
      (?s1 rdf:type sc:Invalid),
      (?s1 sc:violates 'Universal facet of Substantial is not in any State')]

# A Substantial se that is in a State st must (directly or indirectly) be phase of a Substantial su
# that lives the Life that (directly or indirectly) contains st. More succinctly, a Substantial cannot
# be part of the Life of another object.
[validateSubstantialPhaseLife:
   (?se usmo:is_in ?st),
   (?se usmo:is_phase_of ?su),
   (?su usmo:lives ?l),
   noValue(?l usmo:contains ?st) ->
      (?se rdf:type sc:Invalid),
      (?se sc:violates 'Substantial cannot be part of the Life of another object'),
      (?se sc:caused_by ?st)]]

# A Transition cannot lead from an Activity to a State and vice-versa, because States and Activitys
# are Situations at different levels of granularity:
[validateTransitionLevelFollows:
   (?t usmo:follows ?a1),
   (?a1 rdf:type usmo:Activity),
   (?t usmo:precedes ?a2),
   noValue(?a2 rdf:type usmo:Activity) ->
      (?t rdf:type sc:Invalid),
      (?t sc:violates 'Transition cannot lead from Activity to non-Activity'),
      (?t sc:caused_by ?a2)]]

[validateTransitionLevelPrecedes:
   (?t usmo:precedes ?a1),
   (?a1 rdf:type usmo:Activity),
   (?t usmo:follows ?a2),
   noValue(?a2 rdf:type usmo:Activity) ->
      (?t rdf:type sc:Invalid),
      (?t sc:violates 'Transition cannot lead from Activity to non-Activity'),
      (?t sc:caused_by ?a2)]]

# An Event can trigger at most one Transition from the same Situation; in other words, if there
# are several Transitions leading away from a Situation, they must be triggered by different Events.
# Note that the same Event may trigger Transitions from different Situations if it affects several
# objects at the same time:
[validateEventTriggering:
   (?e usmo:triggers ?t1),
   (?t1 rdf:type usmo:Transition),
   (?e usmo:triggers ?t2),
   (?t2 rdf:type usmo:Transition),
   (?t1 owl:sameAs ?t2) ->
      (?e rdf:type sc:Invalid),
      (?e sc:violates 'Event cannot '),
      (?e sc:caused_by ?t2)]]

# If a Behavior b may create a Schema sc, then an Action a instantiating b must actually create a
# Substantial s instantiating sc. A similar rule applies to object destruction:
[validateMayCreate:
   (?b usmo:may_create ?sc),
   (?a rdf:type usmo:Action),
   (?a usmo:is_instance_of ?b),
   notExists(?s usmo:is_instance_of ?sc, ?a usmo:creates ?s) ->
      (?a rdf:type sc:Invalid),
      (?a sc:violates 'Action does not create correctly typed Substantials'),
      (?a sc:caused_by ?s)]]

[validateMayCreate:
   (?b usmo:may_destroy ?sc),
   (?a rdf:type usmo:Action),
   (?a usmo:is_instance_of ?b),
   notExists(?s usmo:is_instance_of ?sc, ?a usmo:destroys ?s) ->
      (?a rdf:type sc:Invalid),
      (?a sc:violates 'Action does not destroy correctly typed Substantials'),
      (?a sc:caused_by ?s)]]

#------------------#
# Deductive rules. #
#------------------#

# If a Set s1 subsets another Set s2, then all members of s1 are also members of s2:
[deriveSubsets:
  (?p usmo:is_member_of ?s1),
      (?s1 usmo:subsets ?s2) ->
      (?p usmo:is_member_of ?s2)]


# If a Schema s1 is subtype of a Schema s2, then the extension of s1 subsets the extension of s2:
[deriveSubsetExtensions:
  (?s1 usmo:is_subtype_of ?s2),
      (?s1 usmo:has_extension ?e1),
      (?s2 usmo:has_extension ?e2) ->
      (?e1 usmo:subsets ?e2)]

# If a Particular is an instance of a Universal, then it is a member of the Universals extension:
[deriveExtensionRelation:
  (?p usmo:is_instance_of ?u),
      (?u usmo:has_extension ?e) ->
      (?p usmo:is_member_of ?e)]

# If a Particular p is an instance of a Schema s1, and s1 is subtype of a Schema s2, then p is also
# an instance of s2:
[deriveSubclassesHasInstance:
  (?p usmo:is_instance_of ?u1),
      (?u1 usmo:is_subtype_of ?u2) ->
      (?p usmo:is_instance_of ?u2)]

# If a Particular p1 is in context of a Situation s, all Particulars p2 that are part of p1 are also in
# the context of s:
[deriveParticularContextParticular:
  (?p1 usmo:has_context ?s),
  (?p1 usmo:contains ?p2) ->
      (?p2 usmo:has_context ?s)]

# If a Particular p is in context of a Situation s1 and s1 is contained in a Situation s2, then p is also
# in the context of s2:
[deriveParticularContextSituation:
  (?p usmo:has_context ?s1),
  (?s1 usmo:is_part_of ?s2) ->
      (?p usmo:has_context ?s2)]

# The Substantials that represent the existential facets of an object instantiate the same Schema
# as the Substantial that represents the universal facet:
[derivePhaseModeling:
  (?s1 usmo:is_phase_of ?s2),
      (?s2 usmo:is_instance_of ?sc) ->
      (?s1 usmo:is_instance_of ?sc)]

# If a Behavior may involve a Property, then it also may involve any Schema that the Property
# depends on. Similarly, if an Action involves a Moment, then it also involves any Substantial that
# the Moment depends on:
[deriveBehaviorInvolvement:
  (?b usmo:may_involve ?p),
      (?p usmo:depends_on ?sc),
      (?sc rdf:type usmo:Schema),
      (?p rdf:type usmo:Property) ->
      (?b usmo:may_involve ?sc)]

[deriveActionInvolvement:
  (?a usmo:involves ?m),
      (?m usmo:depends_on ?s),
      (?m rdf:type usmo:Moment),
      (?s rdf:type usmo:Substantial) ->
      (?a usmo:involves ?s)]

# If a Behavior may involve a Schema, then it also may involve any Entitys that are part of the
# Schema. A similar axiom applies to the involvement of Substantials in Actions.
[deriveBehaviorInvolvementContainedParts:
  (?b usmo:may_involve ?sc),
      (?sc rdf:type usmo:Schema),
      (?sc usmo:contains ?e) ->
      (?b usmo:may_involve ?e)]

[deriveActionInvolvementContainedParts:
  (?a usmo:may_involve ?s),
      (?s rdf:type usmo:Substantial),
      (?s usmo:contains ?e) ->
      (?a usmo:involves ?e)]

# If an Action a involves a Particular p (either a Moment or a Substantial) and a instantiates a
# Behavior b, then b modally involves any Universals u that p is an instance of. Similar axioms
# apply to object creation and destruction:
[deriveBehaviorInvolvementFromAction:
  (?a usmo:involves ?p),
      (?a usmo:is_instance_of ?b),
      (?p usmo:is_instance_of ?u) ->
      (?b usmo:may_involve ?u)]

[deriveBehaviorCreationFromAction:
  (?a usmo:creates ?s),
      (?a usmo:is_instance_of ?b),
      (?s usmo:is_instance_of ?sc) ->
      (?b usmo:may_create ?sc)]

[deriveBehaviorDestroyFromAction:
  (?a usmo:destroys ?s),
      (?a usmo:is_instance_of ?b),
      (?s usmo:is_instance_of ?sc) ->
      (?b usmo:may_destroy ?sc)]
```

Listing A.2: *Jena rule axiomatization of USMO*

Appendix B

Semantic Connector Utility Ontology

B.1 Concepts and Relationships in N3

Listing B.1 shows the semantic connector utility ontology.

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:usmo="http://www.sap.com/research/cgt/usmo#"
    xmlns:protege="http://protege.stanford.edu/plugins/owl/protege#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:p1="http://www.owl-ontologies.com/assert.owl#"
    xmlns="http://www.sap.com/research/cgt/sc#"
  xml:base="http://www.sap.com/research/cgt/sc">
  <owl:Ontology rdf:about="">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >This ontology provides utility classes and properties to be used in the Semantic Connector toolkit.</
          rdfs:comment>
    <owl:imports rdf:resource="http://www.sap.com/research/cgt/usmo"/>
    <owl:versionInfo rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >1.0</owl:versionInfo>
  </owl:Ontology>
  <owl:Class rdf:ID="Remove">
    <rdfs:subClassOf rdf:resource="http://www.sap.com/research/cgt/usmo#Entity"/>
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Classifies all entities that are candidate for removal because they violate some consistency constraint
          or do not reference a model element in the MDSD space.</rdfs:comment>
  </owl:Class>
  <owl:Class rdf:ID="Dangling">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >This class contains entities that do not reference a corresponding model element in the MDSD
          technological space. Membership in this class is asserted via rules.</rdfs:comment>
    <rdfs:subClassOf rdf:resource="#Remove"/>
  </owl:Class>
  <owl:Class rdf:ID="Invalid">
    <rdfs:subClassOf rdf:resource="http://www.sap.com/research/cgt/usmo#Entity"/>
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >This class contains entities that have been inferred to validate certain consistency and integrity
          conditions. Class membership is asserted via appropriate rules written in a rule language.</
          rdfs:comment>
  </owl:Class>
  <owl:ObjectProperty rdf:ID="caused_by">
    <rdfs:domain rdf:resource="#Invalid"/>
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Relates an invalid Entity with those objects that are the reason for the integrity violation.</
          rdfs:comment>
    <rdfs:range rdf:resource="http://www.sap.com/research/cgt/usmo#Entity"/>
  </owl:ObjectProperty>
  <owl:DatatypeProperty rdf:ID="violates">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Associates an invalid entity with a human-readable description of the violation.</rdfs:comment>
    <rdfs:domain rdf:resource="#Invalid"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="maps">
    <rdfs:subPropertyOf>
      <owl:DatatypeProperty rdf:ID="references"/>
    </rdfs:subPropertyOf>
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Links modeling entities. Only one maps relation may exist for one modeling entity.</rdfs:comment>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:about="#references">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Relates an entity in the semantic connector knowledge base with an element in one of the models
          integrated by the connector. This expresses an existential dependency - an ontology individual cannot
           exist without a corresponding model element. Thus, the overall integrity condition for the semantic
          connector is that every individual has an associated model element and these model elements still
          exist in the corresponding models.

This property denotes so-called "soft" references, i.e., that individuals that are linked to the modeling
      technological space via this property were only created as casual mapping action. Corresponding "hard"
      references are linked with the maps property.</rdfs:comment>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="http://www.sap.com/research/cgt/usmo#Entity"/>
  </owl:DatatypeProperty>
</rdf:RDF>

<!-- Created with Protege (with OWL Plugin 3.4, Build 128)  http://protege.stanford.edu -->
```

Listing B.1: *The semantic connector utility ontology in OWL XML syntax*

B.2 Auxiliary Jena Rules

The next listing shows auxiliary rules for the semantic connector utility ontology that resolve dependencies on invalid entities and mark individuals that are to be removed.

```
# Additional helper rules for the semantic connector utility ontology.

@prefix usmo: <http://www.sap.com/research/cgt/usmo#>.
@prefix sc: <http://www.sap.com/research/cgt/sc#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.


# An Entity that depends on an invalid Entity is invalid.
[findDependsOnInvalid:
  (?e1 usmo:depends_on ?e2),(?e2 rdf:type sc:Invalid),
      print('inferInvalidity(invalidEntity,inferredInvalidEntity):',?e2,?e1) ->
   (?e1 rdf:type sc:Invalid),
   (?e1 sc:violates 'Entity depends on an invalid entity'),
   (?e1 sc:caused_by ?e2)]

# An Entity that depends on a dangling Entity is invalid.
[findDependsOnDangling:
  (?e1 usmo:depends_on ?e2),(?e2 rdf:type sc:Dangling),
      print('findDependsOnDangling(dangling entity,inferred invalid entity):',?e2,?e1) ->
    (?e1 rdf:type sc:Invalid),
    (?e1 sc:violates 'Entity depends on an entity that has been deleted from its model'),
    (?e1 sc:caused_by ?e2)]

# Add an explanation to dangling entities
[addExplanationToDangling:
  (?e1 rdf:type sc:Dangling),
      print('addExplanationToDangling(dangling entity):',?e1) ->
    (?e1 sc:violates 'Entity has been removed from its model') ]

[markRemoveDependsOnInvalid:
  (?e1 rdf:type sc:Remove) <-
    (?e1 usmo:depends_on ?e2), (?e2 rdf:type sc:Invalid),
    print('markRemoveDependsOnInvalid (marked entity, causing entity):',?e1,?e2) ]

[markRemoveDependsOnDangling:
  (?e1 usmo:depends_on ?e2), (?e2 rdf:type sc:Dangling),
      print('markRemoveDependsOnDangling (marked entity, causing entity):',?e1,?e2) ->
    (?e1 rdf:type sc:Remove) ]

# An Entity must reference a model element in the MDSD space.
[findDanglingEntities:
   (?e rdf:type usmo:Entity),noValue(?e sc:references ?m),
       print('findDanglingEntities (entity):',?e) ->
      (?e rdf:type sc:Dangling) ]
```

Listing B.2: *Auxiliary semantic connector utility jena rules*

Appendix C

Case Study Generator Templates

The following appendices list excerpts of the generator templates that have been developed during language engineering. Please note that we abandoned complete listings for the sake of brevity.

C.1 Generate Data Code

Listing C.1 shows an excerpt of the Java generator for the data DSL, introduced in Section 7.2.2.1.

```
«IMPORT de_feasiple_salesscenario_businessobject»

«EXTENSION de::feasiple::salesscenario::util::ext::util»
«EXTENSION de_feasiple_salesscenario_businessobject::generator::java::ext::util»

«REM»root template generating the root BusinessBundles object«ENDREM»
«DEFINE bundles FOR BusinessBundles»
        «FOREACH this.bundles AS businessBundle»
                «EXPAND bundle FOR businessBundle-»
        «ENDFOREACH»
        «FOREACH this.types AS businessObject»
                «EXPAND businessObject(this.name) FOR businessObject-»
        «ENDFOREACH»
«ENDDEFINE»


«REM»root template generating a BusinessBundle«ENDREM»
«DEFINE bundle FOR BusinessBundle»
        «FOREACH this.types.typeSelect(BusinessObject) AS businessObject»
                «EXPAND businessObject(this.name) FOR businessObject-»
        «ENDFOREACH»
        «FOREACH this.types.typeSelect(AdvancedType) AS advancedType»
                «EXPAND javaClass FOR advancedType-»
        «ENDFOREACH»
«ENDDEFINE»


«REM»polymophism dummy for next template«ENDREM»
«DEFINE businessObject(String bundleId) FOR DataType»
«ENDDEFINE»


«DEFINE businessObject FOR BusinessArtefact»
«ENDDEFINE»


«REM»extracts BO as BusinessArtefact and each of its items as well«ENDREM»
«DEFINE businessObject(String bundleId) FOR BusinessObject»
«EXPAND javaClass(bundleId) FOR this-»
«EXPAND javaClass(bundleId) FOREACH this.items-»
«ENDDEFINE»


«REM»polymorphism dummy for next template«ENDREM»
«DEFINE javaClass(String bundleId) FOR DataType»
«ENDDEFINE»
```

Listing C.1: *Exerpt of the Java generator for data DSL*

C.2 Generate State Code

Listing C.2 shows an excerpt of the SCXML generator for the state DSL, introduced in Section 7.2.2.2.

```
«IMPORT de_feasiple_salesscenario_state»
«EXTENSION de_feasiple_salesscenario_state::generator::scxml::ext::scxml»
«EXTENSION de::feasiple::salesscenario::util::ext::util»

«DEFINE Root FOR StateMachine-»
«EXPAND Scxml FOR this»
«EXPAND JavaClass FOR this»
«ENDDEFINE»

«DEFINE Scxml FOR StateMachine-»
«FILE getFolderForPackage(this.namespace) + "/" + this.id.toLowerCase() + ".xml"-»
<?xml version="1.0"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0"
        initialstate="«this.startState.id-»">
        «EXPAND scxmlState(this) FOR this.startState»
        «EXPAND scxmlState(this) FOREACH this.states.select(e|e != this.startState) SEPARATOR "\n"-»

        <state id="scxmlEndStateDoNotUseThisStateName" />
</scxml>
«ENDFILE»
«ENDDEFINE»

«DEFINE scxmlState(StateMachine sm) FOR State-»
        <state id="«this.id-»">
                «EXPAND scxmlTransition(sm) FOREACH this.transitions-»
        </state>
«ENDDEFINE»

«DEFINE scxmlAction FOR Action-»
«this.id-»
«ENDDEFINE»

«DEFINE scxmlTransition(StateMachine sm) FOR Transition-»
                <transition event="«sm.id + '.' + this.event»" target="«IF !this.endTransition-»«this.target.
                      id-»«ELSE»scxmlEndStateDoNotUseThisStateName«ENDIF-»" />
«ENDDEFINE»


«DEFINE JavaClass FOR StateMachine-»
«FILE getFileForPackage(this.namespace, this.id.toFirstUpper())»
package «this.namespace-»;

import org.apache.commons.scxml.env.AbstractStateMachine;

public class «this.id.toFirstUpper()-» extends AbstractStateMachine {

«EXPAND JavaEvents FOR this»

        public «this.id.toFirstUpper()-»() {
                super(«this.id.toFirstUpper()-».class.getResource("«this.id.toLowerCase()-».xml"));
        }

        public void fireEvent(Event event) {
                this.fireEvent(event.getId());
        }

        public String getCurrentState() {
                return ((org.apache.commons.scxml.model.State) getEngine()
                                .getCurrentStatus().getStates().iterator().next()).getId();
        }

«EXPAND JavaMethod FOREACH this.states.add(this.startState)-»
}
«ENDFILE»
«ENDDEFINE»

«DEFINE JavaEvents FOR StateMachine-»
        public static enum Event {
        «LET getSet() AS events-»
        «FOREACH this.states.add(this.startState) AS state-»
        «FOREACH state.transitions AS transition-»
        «IF !events.contains(transition.event)-»
                «events.add(transition.event).select(e|e == transition.event).toList().get(0).toUpperCase()-»("
                      «this.id.toFirstLower() + "." + transition.event-»"),
        «ENDIF-»
        «ENDFOREACH-»
        «ENDFOREACH-»
        «ENDLET-»;

                private String id;

                Event(String id) {
                        this.id = id;
                }
```

Listing C.2: *Excerpt of the SCXML generator for state DSL*

C.3 Generate Action Code

Listing C.3 shows an excerpt of the Java generator for the action DSL, introduced in Section 7.2.2.3.

```
«IMPORT de_feasiple_salesscenario_action»
«EXTENSION de::feasiple::salesscenario::util::ext::util»
«EXTENSION de_feasiple_salesscenario_action::generator::java::ext::util»

«DEFINE main FOR ActionBundle»
«FOREACH this.sets AS set»
«EXPAND javaClass(this.name, set.name) FOR set»
«ENDFOREACH»
«FOREACH this.dataTypes AS type»
«EXPAND javaClass(
        this.name,
        typeName(type),
        (SetType.isInstance(type) ? "de_feasiple_salesscenario_action.generator.java.api.AbstractSetParameter<
             Object> " : "de_feasiple_salesscenario_action.generator.java.api.AbstractParameter<Object>"))
        FOR type»
«ENDFOREACH»
«ENDDEFINE»

«DEFINE javaClass(String bundleId, String name) FOR Object»
«EXPAND javaClass(bundleId, name, "") FOR this»
«ENDDEFINE»

«REM»generates a java class from BusinessArtefact«ENDREM»
«DEFINE javaClass(String bundleId, String name, String extends) FOR Object»
«FILE getFileForPackage(bundleId, name)-»
«fileHeader()-»
package «bundleId-»;

public final class «name-» «IF extends != ""-»extends «extends-»«ENDIF-» {
        «EXPAND javaClassContent FOR this-»

}
«ENDFILE»
«ENDDEFINE»

«DEFINE javaClassContent FOR Object»
«ENDDEFINE»

«DEFINE javaClassContent FOR ActionSet»
        «EXPAND javaMethod FOREACH this.actions-»
«ENDDEFINE»


«DEFINE javaMethod FOR Action»
        /**
         *      «this.description»
         *
         **/
        public static

        «IF this.result == null-»
                void
        «ELSE-»
                «typeName(this.result)»
        «ENDIF-»
        «this.name-»(«FOREACH this.arguments AS arg SEPARATOR ','»«typeName(arg)» «arg.name.toFirstLower()»
             Value«ENDFOREACH-») {
        «EXPAND methodContent FOR this-»
        }
«ENDDEFINE»

«DEFINE methodContent FOR Action-»
                // TODO: implement generator
«ENDDEFINE»

«DEFINE methodContent FOR UpdateAction-»
                «PROTECT CSTART "/*" CEND "*/" ID uniqueId(this)»
                // your code goes here
                «ENDPROTECT»
«ENDDEFINE»
```

Listing C.3: *Excerpt of the Java generator for action DSL*

C.4 Generate User Interface Code

Listing C.4 shows an excerpt of the Java SWT generator for the user interface DSL, introduced in Section 7.2.2.4.

```
«EXTENSION de::feasiple::salesscenario::util::ext::util»
«EXTENSION de_feasiple_salesscenario_view::generator::util»
«EXTENSION de_feasiple_salesscenario_view::generator::swt::ext::utils»

«IMPORT de_feasiple_salesscenario_view»

«REM»root template treating the root BusinessBundle«ENDREM»
«DEFINE Root FOR ViewBundle»
        «FOREACH this.dialogs AS dialog-»
        «EXPAND swtFile(this) FOR dialog-»
        «EXPAND generateButtonPanelItemCompositeFiles(this) FOR dialog.content-»
«REM»   «EXPAND generateCompositeUpdaterFiles(this, getCompositeName(dialog)) FOR dialog.content-»«ENDREM»
        «ENDFOREACH»
«ENDDEFINE»

«DEFINE swtFile(ViewBundle bundle) FOR Dialog»
«FILE getFileForPackage(bundle.id, this.id.toFirstUpper() + "Dialog")»
package «bundle.id-»;

import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Listener;

import de_feasiple_salesscenario_view.generator.swt.api.Dialog;

/**
 * The class <code>«this.id.toFirstUpper()-»Dialog</code> ...
 */
public class «this.id.toFirstUpper()-»Dialog extends Dialog<«this.id.toFirstUpper()-»Composite> {

        /**
         * Creates «this.id.toFirstUpper()-»Dialog instance.
         *
         */
        public «this.id.toFirstUpper()-»Dialog(Display display) {
                super(display);
        }

        «EXPAND getSize FOR this-»

        @Override
        protected void createControl() {
                «EXPAND swtElement(bundle, this) FOR this.content-»
                this.shell.setText("«this.title»");
        }

        protected void shellDisposed(/*Transition closingTransition*/) {
                // System.out.println("Closing «getCompositeName(this)-» shell.");
        }

        @Override
        public void update() {
                // implement your UI updates here
        }
}
«ENDFILE»
«ENDDEFINE»

«DEFINE getSize FOR Dialog»
        «IF this.width >= 0-»
        @Override
        protected int getWidth() {
                return «this.width-»;
        }
        «ENDIF-»

        «IF this.height >= 0-»
        @Override
        protected int getHeight() {
                return «this.height-»;
        }
        «ENDIF-»
«ENDDEFINE»


«DEFINE localSWTElement(String swtParent) FOR Widget»
«ENDDEFINE»
«DEFINE swtElement(String swtParent) FOR Widget»
«ENDDEFINE»


«DEFINE swtElement(ViewBundle bundle, Dialog dialog) FOR ContainerWidget-»
        this.control = new «getCompositeName(dialog)-»(this, this.shell, SWT.NONE);
«FILE  getFileForPackage(bundle.id, getCompositeName(dialog))-»
package «bundle.id-»;

import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.custom.TableEditor;
import org.eclipse.swt.graphics.Color;
import org.eclipse.swt.events.MouseEvent;
import org.eclipse.swt.events.MouseListener;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
```

Listing C.4: *Excerpt of the Java SWT generator for UI DSL*

C.5 Generate Context Code

Listing C.5 shows an excerpt of the Java generator for the context DSL, introduced in Section 7.2.2.5.

```
«IMPORT de_feasiple_salesscenario_context»

«EXTENSION de::feasiple::salesscenario::util::ext::util»
«EXTENSION de_feasiple_salesscenario_context::generator::java::ext::ext»

«REM»root template generating the root BusinessBundles object«ENDREM»
«DEFINE root FOR ContextProvider»
        «EXPAND javaContextProvider FOR this-»
        «EXPAND contextInterfaces(this.id) FOREACH this.contexts-»
        «EXPAND javaContextCreator(this.id) FOREACH this.contexts-»
«ENDDEFINE»

«DEFINE contextInterfaces(String bundleId) FOR Context»
«FILE getFileForPackage(bundleId, this.id.toFirstUpper())-»
package «bundleId-»;

import de_feasiple_salesscenario_context.generator.java.api.Contexts;

public interface «this.id.toFirstUpper()-» extends Contexts {
        «EXPAND contextInterfaceDeclaration FOREACH this.children»

        «EXPAND contextInterfaceSlotDeclaration FOREACH this.variables»
}
«ENDFILE»
«ENDDEFINE»


«DEFINE contextInterfaceDeclaration FOR Context»
        interface «this.id.toFirstUpper()-» extends «this.parent.id.toFirstUpper()-» {
                «EXPAND contextInterfaceSlotDeclaration FOREACH this.variables»

                «EXPAND contextInterfaceDeclaration FOREACH this.children»
        }
«ENDDEFINE»


«DEFINE contextInterfaceSlotDeclaration FOR Slot»
                String «this.id.toUpperCase()-» = "«this.id-»";
«ENDDEFINE»


«DEFINE contextInterfaceSlotDeclaration FOR ListSlot»
                String «this.id.toUpperCase()-»_LIST = "«this.id-»";
«ENDDEFINE»


«DEFINE javaContextProvider FOR ContextProvider»
«FILE getFileForPackage(this.id, "ContextProvider")-»
package «this.id-»;

import java.util.Map;

import de_feasiple_salesscenario_context.generator.java.api.AbstractContextProvider;
import de_feasiple_salesscenario_context.generator.java.api.IContextCreator;
import de_feasiple_salesscenario_context.generator.java.api.Contexts;

public final class ContextProvider extends AbstractContextProvider {

        public static final AbstractContextProvider INSTANCE = new ContextProvider();

        private ContextProvider() {
                // singleton
                this.createRootContext(«this.contexts.get(0).id.toFirstUpper()-».class);
        }

        @Override
        protected Map<Class<? extends Contexts>, IContextCreator> fillCreators(
                        Map<Class<? extends Contexts>, IContextCreator> creators) {
                «EXPAND contextProviderRegistrationLine FOREACH this.contexts»
                return creators;
        }
}
«ENDFILE»
«ENDDEFINE»


«DEFINE contextProviderRegistrationLine FOR Context»
                creators.put(«getContextImportPath()-».class,
                                new «this.id.toFirstUpper()-»ContextCreator());
                «EXPAND contextProviderRegistrationLine FOREACH this.children»
«ENDDEFINE»


«DEFINE javaContextCreator(String bundleId) FOR Context»
```

Listing C.5: *Excerpt of the Java generator for context DSL*

C.6 Generate Pageflow Code

Listing C.6 shows an excerpt of the SCXML generator for the pageflow DSL, introduced in Section 7.2.2.6.

```
«IMPORT de_feasiple_salesscenario_dialog»
«EXTENSION de_feasiple_salesscenario_dialog::generator::scxml::ext::scxml»
«EXTENSION de::feasiple::salesscenario::util::ext::util»

«DEFINE Root FOR PageFlowContainer-»
«EXPAND Root(this.namespace) FOR this.rootPageFlow»
«REM»generate global page handling«ENDREM»
«ENDDEFINE»

«DEFINE Root(String namespace) FOR PageFlow-»
«REM»«EXPAND Root(namespace) FOREACH children»«ENDREM»
«EXPAND Scxml(namespace) FOR this»
«EXPAND JavaClass(namespace) FOR this»
«ENDDEFINE»

«DEFINE Scxml(String namespace) FOR PageFlow-»
«FILE getFolderForPackage(namespace) + "/" + getXmlName() + ".xml"-»
<?xml version="1.0"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0"
        initialstate="«getNodeScxmlStateId(this.nodes.typeSelect(StartPage).get(0))-»">
«EXPAND scxmlStates FOR this»
        <state id="«getFinalScxmlStateId()»" />
</scxml>
«ENDFILE»
«ENDDEFINE»

«DEFINE scxmlStates FOR PageFlow-»
        «EXPAND scxmlState(this) FOR this.nodes.typeSelect(StartPage).get(0)»
        «EXPAND scxmlState(this) FOREACH this.nodes.select(e|e != this.nodes.typeSelect(StartPage).get(0))
              SEPARATOR "\n"-»
        «EXPAND scxmlStates FOREACH this.children»
«ENDDEFINE»

«DEFINE scxmlState(PageFlow sm) FOR Node-»
        <state id="«getNodeScxmlStateId()-»">
                «EXPAND scxmlTransition(sm) FOREACH this.transitions-»
        </state>
«ENDDEFINE»

«DEFINE scxmlAction FOR Action-»
«removeSpaces(this.id)-»
«ENDDEFINE»

«DEFINE scxmlTransition(PageFlow sm) FOR Transition-»
        «IF !final -»
                <transition event="«getScxmlTransitionId()»" target="«to.getNodeScxmlStateId()»" />
                «IF this.dispose-»
                <transition event="«getScxmlDisposeTransitionId()»" target="«to.getNodeScxmlStateId()»" />
                «ENDIF-»
        «ELSE-»
                <transition event="«getScxmlTransitionId()»" target="«getFinalScxmlStateId()»" />
                «IF this.dispose-»
                <transition event="«getScxmlDisposeTransitionId()»" target="«getFinalScxmlStateId()»" />
                «ENDIF-»
        «ENDIF-»
«ENDDEFINE»


«DEFINE JavaClass(String namespace) FOR PageFlow-»
«FILE getFileForPackage(namespace, removeSpaces(this.id).toFirstUpper())»
package «namespace-»;

import org.apache.commons.scxml.env.AbstractStateMachine;

public class «getClassName()-» extends AbstractStateMachine {

        public static final «getClassName()» INSTANCE = new «getClassName()»();

«EXPAND JavaEvents FOR this»

        public «getClassName()-»() {
                super(«getClassName()-».class.getResource("«getXmlName()-».xml"));
        }

        public void triggerTransition(Transition transition) {
                this.fireEvent(transition.getId());
        }

«EXPAND JavaMethod FOR this-»

        public void «getFinalScxmlStateId()-»() {
                System.out.println("Exiting. Goodbye.");
                System.exit(0);
        }
```

Listing C.6: *Excerpt of the SCXML generator for pageflow DSL*

List of Figures

List of Tables

List of Listings

List of Abbreviations

4GL	Fourth generation language
ABAP	Advanced Business Application Programming
ABox	Assertional Box
ADL	Architecture Description Language
AE	Application engineering
AM3	ATLAS MegaModel Management
AMMA	ATLAS Model Management Architecture
AMW	ATLAS Model Weaver
ASD	Agile Software Development
ATL	ATLAS Transformation Language
ATP	ATLAS Technical Projectors
BPEL	Web Services Business Process Execution Language
BWW	Bunge-Wand-Weber (ontology)
CRM	Customer Relationship Management
CRUD	Create Retrieve Update Delete
CWA	Closed world assumption
CWM	Common Warehouse Metamodel
DE	Domain engineering
DL	Description Logics
DOC	Distributed Object Computing
DOLCE	Descriptive Ontology for Linguistic and Cognitive Engineering
DSL	Domain-specific language
DSM	Domain-specific model

DSML	Domain-specific modeling language
DSMW	Domain-specific weaving models
DSVML	Domain-specific visual modeling languages
EJB	J2EE Enterprise Java Beans
EMF	Eclipse Modeling Framework
EMOF	Essential Meta Object Facility
ER	Entity Relationship
ERP	Enterprise Resource Planning
FMC	Fundamental Modeling Concepts
FODA	Feature-Oriented Domain Analysis
fUML	Foundational UML Subset
GEMS	Generic Eclipse Modeling System
GFO	General Formal Ontology
GME	Generic Modeling Environment
GMF	Graphical Modeling Framework
GoF	Gang of Four
GOL	General Ontological Language
GOPPRR	Graph-Object-Property-Port-Role-Relationship
GP	Generative Programming
GPL	General purpose programming language
GPML	General purpose modeling language
GTZ	German Technical Cooperation
IDE	Integrated Development Environment
INRIA	National institute for research in informatics and automation
ISC	Invasive Software Composition
ISIS	Institute for Software Integrated Systems
J2EE	Java 2 Enterprise Edition
JBoss AS	JBoss Application Server
jPDL	Java Process Definition Language
JSP	Java Server Pages
KB	Knowledgebase
KM3	Kernel MetaMetaModel

LHS	Left-hand side
M2C	Model-to-code (transformation)
M2M	Model-to-model (transformation)
M2T	Model-to-text (transformation)
MB-UID	Model-Based User Interface Development
MDE	Multi-Domain Engineering
MDSD	Model-driven software development
MOF	Meta Object Facility
MOIN	Modeling Infrastructure
MPP	Marketing and product plans
MRD	Market requirements document
N3	Notation 3
oAW	openArchitectureWare
OCL	Object Constraint Language
ODE	Ontology-based Domain Engineering
OFBiz	Apache Open for Business Project
OO	Object-orientated programming
OOPP	Objectives-oriented Project Planning
OS	Operating system
OSGi	*former* Open Service Gateway Initiative
OWA	Open world assumption
OWL	Web Ontology Language
PLM	Product Life Cycle Management
PRD	Product requirements document
QVT	Query/View/Transformations
RDFS	RDF Schema
RFID	Radio Frequency Identification
RFP	Request for proposals
RHS	Right-hand side
RuleML	Ruke Markup Language
SCM	Supply Chain Management
SCXML	State Chart XML

SDD	Software design description
Sem-MT-Tool	semantic-enabled model transformation tool
SemIDE	semantic-enabled IDE
seSED	semantic-enabled Software Engineering and Development
SPARQL	SPARQL Protocol and RDF Query Language
SPLE	Software product line engineering
SQL	Structured Query Language
SRM	Supplier Relationship Management
SRS	Software requirements specification
SWRL	Semantic Web Rule Language
SWT	Standard Widget Toolkit
TBox	Terminological Box
TCO	Total Cost of Ownership
TS	Technological space
UI	User interface
UML	Unified Modeling Language
UNA	Unique name assumption
URI	Unified Resource Identifier
USMO	Unified Software Modeling Ontology
VIM	Variant Independent Model
VSM	Variant Specific Model
W3C	World Wide Web Consortium
WYSIWYG	What You See Is What You Get
XAML	eXtensible Application Markup Language
XML	Extensible Markup Language
XP	Extreme Programming
XPDL	XML Process Definition Language
XUL	XML User Interface Language
xUML	Executable UML
ZOPP	Zielorientierte Projektplanung

Bibliography

[AAH05] Petteri Alahuhta, Jari Ahola, and Hannu Hakala. Mobilizing Business Applications. Technical report, National Technology Agency of Finland (Tekes), 2005. 34

[ABBJ06] Freddy Allilaire, Jean Bézivin, Hugo Brunelière, and Frédéric Jouault. Global Model Management in Eclipse GMT/AM3. In *Proceedings of the Eclipse Technology eXchange workshop (eTX) at the ECOOP 2006 Conference*, Nantes, France, 2006. 11

[Adu09] Aduna. openRDF.org ...the home of Sesame. `http://www.openrdf.org/`, 2009. 134

[AE02] J. Ø. Aagedal and E. Ecklund. Modelling QoS: Towards a UML Profile. In *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*, number 2460 in LNCS, pages 275–289, October 2002. 67

[AG09] SAP AG. SAP - Business Software Solutions Applications and Services. `http://www.sap.com`, 2009. 50

[AK03] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003. 5, 8, 12, 60

[All09] OSGi Alliance. OSGi Alliance Specifications. `http://www.osgi.org/Specifications/HomePage?section=2`, September 2009. 139

[Amb02] Scott W. Ambler. Toward Executable UML. *Dr. Dobb's Journal*, January 2002. 67

[Ant02] Grigoris Antoniou. A nonmonotonic rule system using ontologies. In *Proceedings of the International Workshop on Rule Markup Languages for Business Rules on the Semantic Web*, 2002. 29

[Ant07] Michal Antkiewicz. Round-Trip Engineering Using Framework-Specific Modeling Languages. In *Proceedings of OOPSLA07*, Montréal, Québec, Canada., October 2007. 6

[Aßm03] Uwe Aßmann. *Invasive Software Composition*. Springer-Verlag New York Inc., 2003. 100, 101, 102, 103

[AvH04] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. Cooperative Information Systems. The MIT Press, Cambridge, Massachusetts, 2004. 28, 29

[AZW06] Uwe A{ss6, 8, 120

mann, Steffen Zschaler, and Gerd Wagner. *Ontologies, Meta-models, and the Model-Driven Paradigm*, chapter Ontologies, Meta-models, and the Model-Driven Paradigm, pages 249–273. Springer Berlin Heidelberg, 2006.

[Bar07] Neil Bartlett. A Comparison of Eclipse Extensions and OSGi Services. *Eclipse Zone*, February 2007. Publicly Available under http://www.eclipsezone.com/articles/extensions-vs-services/. 139

[Bat05] Ferenc Bator. Virtuelle Maschinen. Technical report, Devmatic IT, 2005. Virtuelle Maschinen sind heute nicht mehr wegzudenken. Sie werden heute auf Mainframes, Desktop Computern und auch Handys und PDA's verwendet. Dieser Artikel beschreibt die am weitesten verbreitesten Virtuellen Maschinen, die Prozess VM's. Prominente Vertreter sind die Java VM und die .NET CLR. 13

[BBB+] Kent Beck, Mike Beedle, Arie Bennekum, Alistair Cockburn, Ward Cunningham, and Martin Fowler. Manifesto for Agile Software Development. http://agilemanifesto.org/. 110

[BCM+07] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2nd edition, August 2007. 23, 24, 28

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1st edition, October 1999. 110

[Bec04] Dave Beckett. *RDF/XML Syntax Specification (Revised)*. World Wide Web Consortium (W3C), February 2004. W3C Recommendation, http://www.w3.org/TR/rdf-syntax-grammar/. 26

[Bet02] Jorn Bettin. Measuring the potential of domain-specific modelling techniques. 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. SoftMetaWare Ltd. 45

[Bet04] Jorn Bettin. *Model-Driven Software Development Activities: The Process View of an MDSD Project*. SoftMetaWare Ltd., May 2004. http://www.softmetaware.com/mdsd-process.pdf. 5, 111, 113, 119

[BET08] E. Biermann, C. Ermel, and G. Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In K. Czarnecki, editor, *Proc. ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS'08)*, volume 5301, pages 53–67, 2008. 158

[BEZ04] Rüdiger Buck-Emden and Peter Zencke. *mySAP CRM: The Official Guidebook to SAP CRM 4.0*. Galileo Press, 2004. 154, 155

[BG04] Dan Brickley and R.V. Guha. *RDF Vocabulary Description Language 1.0: RDF Schema*. World Wide Web Consortium (W3C), February 2004. W3C Recommendation, http://www.w3.org/TR/rdf-schema/. 26, 52

[BH01] Jan Bosch and Mattias Högström. Product Instantiation in Software Product Lines: A Case Study. In *GCSE '00: Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers*, pages 147–162, London, UK, 2001. Springer-Verlag. 22

[BH03] P. Burek and H Herre. Conceptual Modeling based on Upper-Level Ontologies and Meta-Ontological Foundations. In K.-P. Fähnrich and H. Herre, editors, *Content- und Wissensmanagement: Arbeiten aus dem Forschungsvorhaben PreBIS und Beiträge auf den Leipziger Informatik-Tagen 2003*, volume 1 of *Leipziger Beiträge zur Informatik*, pages 153–157. Leipziger Informatik Verbund (LIV), Leipzig, 2003. 77

[BL00] Tim Berners-Lee. Semantic Web - XML2000. Presentation Slides: http://www.w3.org/2000/Talks/1206-xml2k-tbl, 2000. 25, 219

[BL06] Tim Berners-Lee. *Notation 3*. World Wide Web Consortium (W3C), March 2006. Publicly available under http://www.w3.org/DesignIssues/Notation3.html. 197

[BL07] Matthias Bräuer and Henrik Lochmann. Towards Semantic Integration of Multiple Domain-Specific Languages Using Ontological Foundations. In *Proceedings of 4th International Workshop on (Software) Language Engineering (ATEM'07) co-located with the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007), Nashville, TN, USA*, October 2007. 53, 79, 206

[BL08] Matthias Bräuer and Henrik Lochmann. An Ontology for Software Models and its Practical Implications for Semantic Web Reasoning. In *Proceedings of 5th European Semantic Web Conference*, 2008. 53, 79, 191

[BLFM05] Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. Network Working Group, W3C, January 2005. 25

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web: A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American*, May 2001. 25

[Blo08] Joshua Bloch. *Effective Java, 2nd Edition*. Java Series. Prentice Hall, 2nd edition, May 2008. 18

[Blu08] Bluetooth Special Interest Group (SIG). Bluetooth.org. `http://www.bluetooth.org`, 2008. 34

[Boe86] B. Boehm. A spiral model of software development and enhancement. *SIGSOFT Softw. Eng. Notes*, 11(4):14–24, 1986. 110

[BPE07] *Web Services Business Process Execution Language Version 2.0*, April 2007. OASIS Standard, `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf`. 84

[BPM06] Object Management Group (OMG). *Business Process Modeling Notation (BPMN) Specification*, February 2006. Final Adopted Specification, OMG document dtc/06-02-01, `http://www.omg.org/docs/dtc/06-02-01.pdf`. 84

[BPSM$^+$08] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. World Wide Web Consortium (W3C), November 2008. W3C Recommendation, `http://www.w3.org/TR/2008/REC-xml-20081126/`. 25

[BR06] Bernhard Bauer and Stephan Roser. Semantic-enabled Software Engineering and Development. *INFORMATIK 2006 - Informatik für Menschen, Lecture Notes in Informatics (LNI)*, P-94:293–296, 2006. 74, 75

[Brä07a] Matthias Bräuer. Design and Prototypical Implementation of a Pivot Model as Exchange Format for Models and Metamodels in a QVT/OCL Development Environment. Großer Beleg, Technische Universität Dresden, May 2007. http://dresden-ocl.sourceforge.net/gbbraeuer/. 74

[Brä07b] Matthias Bräuer. Design of a Semantic Connector Model for Composition of Metamodels in the Context of Software Variability. Diplomarbeit, Technische Universität Dresden, Department of Computer Science, November 2007. 24, 79, 82, 85, 91, 219, 220, 223

[BS85] Ronald J. Brachman and James G. Schmolze. An overview of the kl-one knowledge representation system. *Cognitive Science*, 9(2):171 – 216, 1985. 28

[BSM$^+$04] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework: a Developer's Guide*. Addison-Wesley, 2004. 8, 50

[BT75] Victor R. Basili and Albert J. Turner. Iterative enhancement: A practical technique for software development. *IEEE Trans. Software Eng.*, 1(4):390–396, 1975. 110

[Bun77] Mario Bunge. *Treatise On Basic Philosophy: Volume 3: Ontology I: The Furniture of the World*. Reidel Publishing Co., Dordrecht, 1977. 85

[Bun79] Mario Bunge. *Treatise On Basic Philosophy: Volume 4: Ontology II: A World of Systems*. Reidel Publishing Co., Dordrecht, 1979. 85

[BvHH$^+$04] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. *OWL Web Ontology Language Reference*. World Wide Web Consortium (W3C), February 2004. W3C Recommendation, `http://www.w3.org/TR/owl-ref/`. 125

[CA05] Krzysztof Czarnecki and MichałAntkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proceedings of Fourth International Conference on Generative Programming and Component Engineering (GPCE'05)*, volume 3676 of *LNCS*, Berlin Heidelberg, September 2005. Springer. 39

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools* *and Applications*. Addison-Wesley Longman, Amsterdam, 2000. 12, 13, 21, 99, 111

[CEK03] Tony Clark, Andy Evans, and Stuart Kent. Aspect-oriented Metamodelling. *The Computer Journal, 46(5),c British Computer Society 2003; all rights reserved*, 2003. 102

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *Online proceedings of the 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA*, 2003. 8, 9, 10, 11

[Che76] Peter Pin-Shan Chen. The Entity Relationship Model — Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976. 83

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. In *Software Process Improvement and Practice, special issue on "Software Variability: Process and Management"*, volume 10(2), pages 143 – 169, 2005. 20

[CM03] W.F. Clocksin and C.S. Mellish. *Programming in Prolog. Using the ISO Standard*. 2003. 76

[Coc02] Alistair Cockburn. Use cases, ten years later. *STQE magazine*, April 2002. Publicly available under http://alistair.cockburn.us/Use+cases,+ten+years+later. 112

[Con07a] AMPLE Consortium. AMPLE Research Project. www.ample-project.net, Dec 2007. 39, 133, 153

[Con07b] EMODE Consortium. Emode: Enabling model transformation-based cost efficient adaptive multi-modal user interfaces. http://www.emode-projekt.de/, 2007. Corresponding information available under http://www.emode-projekt.de/. 85

[Con08] The Eclipse Consortium. Eclipse Roadmap v4. http://www.eclipse.org/org/councils/roadmap_v4_0, December 2008. 139

[Coo00] Steve Cook. The UML family: Profiles, Prefaces and Packages. In Andy Evans, Stuart Kent, and Bran Selic, editors, *Proceedings of the Third International Conference on The Unified Modeling Language – Advancing the Standard (UML 2000), York, UK*, volume 1939 of *LNCS*, pages 255–264. Springer, October 2000. 15

[Coo06] Steve Cook. Domain-Specific Modeling. *The Architecture Journal*, 9:10–17, 2006. http://msdn.microsoft.com/en-us/library/bb245773.aspx. 15, 111

[Cor09a] Microsoft Corporation. "intellipad" overview ("intellipad"). http://msdn.microsoft.com/de-de/library/dd861735(en-us,VS.85).aspx, 2009. Prerelease Documentation. 12

[Cor09b] Microsoft Corporation. Microsoft silverlight. http://silverlight.net/, 2009. The official Microsoft Silverlight website. 84

[CP09] LLC Clark & Parsia. Pellet: The Open Source OWL DL Reasoner. http://clarkparsia.com/pellet, 2009. 28, 52

[CR06] Eric Clayberg and Dan Rubel. *eclipse - Building Commercial-Quality Plug.ins*. Addison Wesley, 2nd edition, 2006. 139

[CvHH$^+$01] Dan Connolly, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. *DAML+OIL (March 2001) Reference Description*. World Wide Web Consortium (W3C), December 2001. W3C Note, http://www.w3.org/TR/daml+oil-reference. 26

[CWM03] Object Management Group (OMG). *Common Warehouse Metamodel (CWM) Specification, Version 1.1, Volume 1*, March 2003. OMG document formal/03-03-02, http://www.omg.org/docs/formal/03-03-02.pdf. 7, 83

[Cza05] Krzysztof Czarnecki. Overview of Generative Software Development. In *UPP05*, 2005. 19, 21, 111, 113

[dAFGD02] Ricardo de Almeida Falbo, Giancarlo Guizzardi, and Katia Cristina Duarte. An ontological approach to domain engineering. In *In Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE02)*, page 351358. ACM, 2002. 117

[Dal09] Mark Dalgarno. Code Generation Network. http://www.codegeneration.net, 2009. 10

[dH09] Johan den Haan. DSL development: 7 recommendations for Domain Specific Language design based on Domain-Driven Design. *The Enterprise Architect*, May 2009. Publicly available under http://www.theenterprisearchitect.eu/archive/2009/05/06/dsl-development-7-recommendations-for-domain-specific-language-\design-based-on-domain-driven-design. 117

[DHHS01] Wolfgang Degen, Barbara Heller, Heinrich Herre, and Barry Smith. GOL: A General Ontological Language. In *Proceedings of the International Conference on Formal Ontology in Information Systems - Volume 2001, Ogunquit, Maine, USA*. ACM Press, October 2001. 77, 87

[DHL03] Martin Doerr, Jane Hunter, and Carl Lagoze. Towards a Core Ontology for Information Integration. *Journal of Digital Information (JoDI)*, 4(1), 2003. Article No. 169. 81, 89

[dNR04] Dionisio de Niz and Raj Rajkumar. Glue code generation: Closing the loophole in model-based development. In *Proceedings of 2nd RTAS Workshop on Model-Driven Embedded Systems (MoDES '04)*, May 2004. 100

[Ecl] Eclipse.org. Eclipse Modeling Framework (EMF) Project. http://www.eclipse.org/modeling/emf. 134

[Ecl09] Eclipse - An Open Development Platform. http://www.eclipse.org, 2009. 10, 11

[EFH$^+$08] Sven Efftinge, Peter Friese, Arno Haase, Dennis Hübner, Clemens Kadura, Bernd Kolb, Jan Köhnlein, Dieter Moroff, Karsten Thoms, Markus Völter, Patrick Schönbach, Moritz Eysholdt, Dennis Hübner, and Steven Reinisch. *openArchitectureWare User Guide Version 4.3.1*, December 2008. Publicly available under http://www.openarchitectureware.org/pub/documentation/4.3.1/openArchitectureWare-4.3.1-Reference.pdf. 164

[EJ01] Robert Esser and Jörn W. Janneck. A Framework for Defining Domain-Specific Visual Languages. In *Proceedings of OOPSLA 2001 Workshop on Domain-Specific Visual Languages*, Tampa Bay, Florida, USA, October 2001. Jyväskylä University Printing House. 111

[EW01] Joerg Evermann and Yair Wand. Towards Ontologically Based Semantics for UML Constructs. In Hideko S. Kunii, Sushil Jajodia, and Arne Sølvberg, editors, *Proceedings ER 2001. 20th International Conference on Conceptual Modeling, Yokohama, Japan*, volume 2224 of *LNCS*, pages 354–367, Berlin Heidelberg, 2001. Springer. 77, 90

[EW05] Joerg Evermann and Yair Wand. Ontology based object-oriented domain modelling: fundamental concepts. *Requirements Engineering*, 10(2):146–160, May 2005. 85, 90

[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. 11, 73

[FBJV05] Marcos Didonet Del Fabro, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. Applying Generic Model Management to Data Mapping. In *Proceedings of the Journées Bases de Données Avancées (BDA05)*, 2005. 73

[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. 73

[FC05] Jim Farley and William Crawford. *Java Enterprise in a Nutshell.* O' Reilly & Associates, Inc., 2005. 69

[Fea07] feasiPLe - Feature-driven, aspect-oriented and model-driven Software Product Line Development. http://www.feasiple.de/index_en.html, 2007. 39, 127, 133, 153

[FGB04] Shayne Flint, Henry Gardner, and Clive Boughton. Executable/Translatable UML in Computing Education. In Raymond Lister and Alison L. Young, editors, *Sixth Australasian Computing Education Conference (ACE2004)*, volume 30 of *CRPIT*, pages 69–75, Dunedin, New Zealand, 2004. ACS. 66

[FGH$^+$94] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency Handling in Multi-Perspective Specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, August 1994. 16

[Fin06] Klaus Finkenzeller. *RFID-Handbuch: Grundlagen und praktische Anwendungen induktiver Funkanlagen, Transponder und kontaktloser Chipkarten.* Hanser Fachbuchverlag, August 2006. 34

[FKN$^+$92] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–58, March 1992. 16

[FNTZ98] T. Fischer, Jörg Niere, L. Torunski, and Albert Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In G. Engels and G. Rozenberg, editors, *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764, pages 296–309. Springer Verlag, November 1998. 84

[Fou] The Eclipse Foundation. SWT: The Standard Widget Toolkit. `http://www.eclipse.org/swt/`. Official website `http://www.eclipse.org/swt/`. 166

[Fou09a] Apache Software Foundation. The Apache Software Foundation. `http://apache.org/`, April 2009. 69

[Fou09b] The Apache Foundation. Apache Struts. `http://struts.apache.org/`, April 2009. 69

[Fou09c] The Apache Software Foundation. Commons SCXML. `http://commons.apache.org/scxml/`, January 2009. 44, 161

[Fou09d] The Apache Software Foundation. The Apache Open For Business Project. `http://ofbiz.apache.org/`, 2009. 16, 17, 70

[Fou09e] The Eclipse Foundation. Graphical Modeling Framework. http://www.eclipse.org/modeling/gmf/, 2009. 11, 56

[FTSDT09] University of Paderborn Fujaba Tool Suite Developer Team. Fujaba Tool Suite. `http://www.fujaba.de/`, 2009. 67, 84

[FV07] Marcos Didonet Del Fabro and Patrick Valduriez. Semi-automatic Model Integration using Matching Transformations and Weaving Models. In *The 22th Annual ACM Symposium on Applied Computing - Model Transformation Track (MT 2007), Seoul, Korea*, pages 963–970, 2007. 73

[FW04] David C. Fallside and Priscilla Walmsley. *XML Schema Part 0: Primer Second Edition.* W3C, October 2004. 25

[GFO09] GFO OWL version. `http://www.onto-med.de/ontologies/gfo.owl`, July 2009. 90

[GG95] Nicola Guarino and Pierdaniele Giaretta. Ontologies and Knowledge Bases: Towards a Terminological Clarification. In N.J.I. Mars, editor, *Towards Very Large Knowledge Bases*, Knowledge Building & Knowledge Sharing. IOS Press, Amsterdam, The Netherlands, 1995. 22, 23

[GG06] Nicola Guarino and Giancarlo Guizzardi. In the Defense of Ontological Foundations for Conceptual Modeling. Published in the Scandinavian Journal of Information Systems Debate Forum, in reply to the article entitled "On Ontological Foundations of Conceptual Modeling" by B. Wyssusek, 2006. 78, 85

[GGM$^+$02] Aldo Gangemi, Nicola Guarino, Claudio Masolo, Alessandro Oltramari, and Luc Schneider. Sweetening Ontologies with DOLCE. In *Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW'02)*, Madrid, Spain, October 2002. Springer. 88

[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. 100

[GHS95] Dimitrios Georgakopoulos, Mark F. Hornick, and Amit P. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995. 84

[GHW02a] Giancarlo Guizzardi, Heinrich Herre, and Gerd Wagner. On the General Ontological Foundations of Conceptual Modeling. In *Proceedings of 21th International Conference on Conceptual Modeling (ER 2002)*, LNCS 2503, Berlin, 2002. Springer. 77, 85, 87, 90, 220

[GHW02b] Giancarlo Guizzardi, Heinrich Herre, and Gerd Wagner. Towards Ontological Foundations for UML Conceptual Models. In R. Meersmann and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, ODBASE. Proceedings of the Confederated International Conferences, Irvine, California*, volume 2519 of *LNCS*, pages 1100–1117, Berlin, 2002. Springer. 77, 90

[GL96] Giuseppe De Giacomo and Maurlzio Lenzerinl. Tbox and abox reasoning in expressive description logics. Technical report, Dipartimento di Informatica e Sistemistica Universit& di Roma "La Sapienza" Via Salaria 113, 00198 Roma, Italy, 1996. 23

[GN87] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, Los Altos, California, 1987. 22, 23

[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, Vancouver, Canada, October 2004. University of Jyväskylä. 111

[GPvS02] Giancarlo Guizzardi, Luís 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. 15, 117

[GPvS05] Giancarlo Guizzardi, Luís Ferreira Pires, and Marten van Sinderen. Ontology-Based Evaluation and Design of Domain-Specific Visual Modeling Languages. In *Proceedings of the 14th International Conference on Information Systems Development, Karlstad, Sweden*, 2005. 111

[Gre06] Adam Greenfield. *Everyware: The Dawning Age of Ubiquitous Computing.* New Riders Publishing, 1st edition, March 2006. 34

[Gro05] Object Management Group. *Semantics of a Foundational Subset for Executable UML Models (Request For Proposal).* Object Management Group, First Needham Place 250 First Avenue, Suite 100 Needham, MA 02494, April 2005. OMG Document: ad/2005-04-02, http://www.omg.org/docs/ad/05-04-02.pdf. 66

[Gro08] Object Management Group. *Semantics of a Foundational Subset for Executable UML Models (FTF - Beta 1).* Object Management Group, beta 1 edition, November 2008. OMG Document Number: ptc/2008-11-03, http://www.omg.org/docs/ptc/08-11-03.pdf. 66

[Gru95] Thomas Gruber. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *International Journal Human-Computer Studies*, 43(5–6):907–928, November 1995. 22, 23

[GS04] Jack Greenfield and Keith Short. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley & Sons, Indianapolis, Indiana, USA, 2004. 6, 7, 8, 16, 67, 68, 82, 99, 220

[Gua98] Nicola Guarino. Formal Ontology and Information Systems. In *Proceedings of FOIS'98, Trento, Italy, 6-8 June 1998*, pages 3–15, Amsterdam, 1998. IOS Press. 23, 85

[Gui05] Giancarlo Guizzardi. *Ontological Foundations for Structural Conceptual Models.* PhD thesis, University of Twente, Enschede, The Netherlands, 2005. Telematica Instituut Fundamental Research Series No. 15. 22, 23, 24, 61

[Gui07] Giancarlo Guizzardi. On Ontology, ontologies, Conceptualizations, Modeling Languages, and (Meta)Models. *Frontiers in Artificial Intelligence and Applications, Databases and Information Systems IV*, 2007. 6, 7, 23, 55, 62, 120

[GW05] Giancarlo Guizzardi and Gerd Wagner. *Applications of a Unified Foundational Ontology*, chapter XIII: Some Applications of a Unified Foundational Ontology in Business Modeling, pages 345–367. IDEA Publisher, 2005. 88, 89, 220

[GWGvS04] Giancarlo Guizzardi, Gerd Wagner, Nicola Guarino, and Marten van Sinderen. An Ontologically Well-Founded Profile for UML Conceptual Models. In *Proceedings 16th International Conference on Advances in Information Systems Engineering (CAiSE), Latvia*, LNCS, page 3084, Berlin, 2004. Springer. 89, 90

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, pages 231–274, August 1987. Elsevier Science Publishers, North-Holland. 84

[Har02] The Harmony project. `http://metadata.net/harmony`, June 2002. 81

[HCW07] Anders Hessellund, Krzysztof Czarnecki, and Andrzej Wasowski. Guided Development with Multiple Domain-Specific Languages. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *ACM/IEEE 10th International Conference On Model Driven Engineering Languages and Systems (MoDELS 2007), Nashville, TN, USA*, volume 4735 of *LNCS*. Springer, September/October 2007. 52, 70, 76, 77, 187, 220, 223

[Her08] Christian Hermann. A Composition Framework for Semantic-Based Generator Code in Multi-Domain Software Development. Diplomarbeit, Technische Universität Dresden, Department of Computer Science, June 2008. 79

[Hes09] Anders Hessellund. The SmartEMF Project. `http://sourceforge.net/projects/smartemf`, 2009. 76

[HG97] Stefan Helming and Michael Göbel. *ZOPP Objectives-oriented Project Planning.* Deutsche Gesellschaft für Technische Zusammenarbeit (GTZ) GmbH, 1997. 36

[HH03a] Barbara Heller and Heinrich Herre. Formal Ontology and Principles of GOL. Technical Report 1, Institute for Medical Informatics, Statistics and Epidemiology (IMISE), Institute for Informatics (IfI), Department of Formal Concepts, University of Leipzig, Germany, July 2003. 87

[HH03b] Barbara Heller and Heinrich Herre. Ontological Categories in GOL. In J. Seibt, editor, *Process Theories: Crossdisciplinary Studies in Dynamic Categories*, pages 57–77. Kluwer Academic Publishers, Dordrecht, 2003. 61

[HHB$^+$06] Heinrich Herre, Barbara Heller, Patryk Burek, Robert Hoehndorf, Frank Loebe, and Hannes Michalek. General Formal Ontology (GFO), Part I: Basic Principles, Version 1.0. OntoMed Report 8, Institute of Medical Informatics, Statistics and Epidemiology (IMISE), Institute of Informatics (IfI), Department of Formal Concepts, University of Leipzig, July 2006. 61

[HJK$^+$09] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual Syntax for Models. In *Proceedings of the 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*, 2009. 119

[HKR$^+$04] Matthew Horridge, Holger Knublauch, Alan Rector, Robert Stevens, and Chris Wroe. *A Practical Guide To Building OWL Ontologies Using The Protégé-OWL Plugin and CO-ODE Tools Edition 1.0.* The University Of Manchester, August 2004. 121

[HKW08] Florian Heidenreich, Jan Kopcsek, and Christian Wende. FeatureMapper: Mapping Features to Models. In *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 943–944, New York, NY, USA, May 2008. ACM. 39, 130

[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 (AOPLE'06) co-located with the International Conference on Generative Programming and Component Engineering (GPCE'06)*, Portland, Oregon, USA, October 2006. 22, 39, 127

[HLL02] Jane Hunter, Carl Lagoze, and Suzanne Little. ABC OWL Schema. http://metadata.net/harmony/ABC/ABC.owl, 2002. 81

[Hol95] David Hollingsworth. *The Workflow Reference Model, Version 1.1*. Workflow Management Coalition, Winchester, Hampshire, UK, January 1995. http://www.wfmc.org/standards/docs/tc003v11.pdf. 84

[HPSB+04] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. World Wide Web Consortium (W3C), May 2004. W3C Member Submission, http://www.w3.org/Submission/SWRL/. 29

[Hru05] Pavel Hruby. Domain-Driven Development with Ontologies and Aspects. In Juha-Pekka Tolvanen, Jonathan Sprinkle, and Matti Rossi, editors, *Proceedings of the 5th OOPSLA Workshop on Domain-Specific Modeling (DSM05)*, number TR-36 in Computer Science and Information System Reports, Technical Reports. University of Jyväskylä, Finland, 2005. 117

[HS08] Anders Hessellund and Peter Sestoft. Flow Analysis of Code Customizations. In Jan Vitek, editor, *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus*, volume 5142 of *LNCS*. Springer, July 2008. 71

[hT09] The hibernate.org Team. Hibernate - Relational Persistence for Java and .NET. http://www.hibernate.org, 2009. 69

[Hun03] Jane Hunter. Enhancing the Semantic Interoperability of Multimedia through a Core Ontology. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(1):49–58, 2003. 81, 89

[HW08] Anders Hessellund and Andrzej Wąsowski. Interfaces and Metainterfaces for Models and Metamodels. In Krzysztof Czarnecki, editor, *ACM/IEEE 11th International Conference On Model Driven Engineering Languages and Systems (MoDELS 2008), Toulouse, France*, 2008. to appear. 71

[IBM09] IBM. IBM Rational Software Architect. http://www-01.ibm.com/software/awdtools/swarchitect/standard/, 2009. 67

[Inc09] Adobe Systems Incorporated. Adobe flex 3. http://www.adobe.com/de/products/flex/, 2009. The offical Adobe Flex website. 84

[INR09] INRIA and LINA. AMMA – ATLAS Model Management Architecture. http://www.sciences.univ-nantes.fr/lina/atl, 2009. 11, 50

[JB06] Frédéric Jouault and Jean Bézivin. 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, May 2006. Springer Berlin / Heidelberg. 11, 50

[JBC+06] Frédéric Jouault, Jean Bézivin, Charles Consel, Ivan Kurtev, and Fabien Latry. Building DSLs with AMMA/ATL: a Case Study on SPL and CPL Telephony Languages. In *1st ECOOP Workshop on Domain-Specific Program Development (DSPD), in conjunction with ECOOP 2006*, Nantes, France, July 2006. 111

[JBo08] JBoss Enterprise Application Platform. http://www.jboss.com/products/platforms/application, 2008. 13, 17, 69

[JBP08] JBoss jBPM. http://jboss.com/products/jbpm, 2008. 17

[JBS08] JBoss Seam. http://www.jboss.com/products/seam, 2008. 69, 156

[Jen09] Jena – A Semantic Web Framework for Java. http://jena.sourceforge.net, January 2009. Version 2.5. 29, 52, 134, 136

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica*, 2005. 11

[Joh05] Rod Johnson. J2EE Development Frameworks. *Computer*, 38(1):107–110, 2005. 69

[Joh07] Rod Johnson. Introduction to the Spring Framework. http://www.theserverside.com/tt/articles/article.tss?l=IntrotoSpring25, October 2007. 17, 156

[Jon04] David E. Jones. *The Open For Business Project: Mini-Language Guide*, August 2004. 70

[JV09] Johannes Kepler University of Linz and Vienna University of Technology. ModelCVS: A Semantic Infrastructure for Model-based Tool Integration. http://www.modelcvs.org, 2009. 75

[KAB02] Ivan Kurtev, Mehmet Aksit, and Jean Bézivin. A Global Perspective on Technological Spaces. Available at: http://wwwhome.cs.utwente.nl/~kurtev/TechnologicalSpaces.doc, 2002. 53, 74

[KAO09] KAON2 - Ontology Management for the Semantic Web. http://kaon2.semanticweb.org/, 2009. 28

[KC04] Graham Klyne and Jeremy J. Carroll. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. World Wide Web Consortium (W3C), February 2004. W3C Recommendation, http://www.w3.org/TR/rdf-concepts/. 26, 27, 52

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1990. 20

[Kel05] Horst Keller. *The Official ABAP Reference*. SAP Press, 2005. 15

[Kel07] Steven Kelly. Domain-Specific Modelling: The Killer App for Method Engineering? In *Proceedings of IFIP WG8.1 working conference on Situational Method Engineering*, 2007. Keynote. 119

[KKK+05] Gerti Kappel, Gerhard Kramler, Elisabeth Kapsammer, Thomas Reiter, Werner Retschitzegger, and Wieland Schwinger. ModelCVS - A Semantic Infrastructure for Model-based Tool Integration. 2005. 120

[KKK+06] Gerti Kappel, Elisabeth Kapsammer, Horst Kargl, Gerhard Kramler, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. On Models and Ontologies - A Layered Approach for Model-based Tool Integration. In H. C. Mayr and R. Breu, editors, *Proceedings of Modellierung 2006, GI-Edition, Lecture Notes in Informatics, Innsbruck, Austria, 22-24 March*, 2006. 75, 76, 220

[KKR+06] G. Kramler, G. Kappel, T. Reiter, E. Kapsammer, W. Retschitzegger, and W. Schwinger. Towards a Semantic Infrastructure Supporting Model-based Tool Integration. In *International Conference on Software Engineering, Proceedings of the 2006 International Workshop on Global Integrated Model Management, Shanghai, China, Session Metamodels and Semantics*, pages 43–46, New York, NY, USA, 2006. ACM Press. 75

[KLD02] Kyo C. Kang, Jaejoon Lee, and Patrick Donohoe. Feature-Oriented Product Line Engineering. *IEEE Software*, 19:58–65, July/August 2002. 19, 21, 22, 219

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, Berlin Heidelberg, June 1997. Springer. 102

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the Association for Computing Machinery*, 42(4):741–843, 1995. 134

[KMR+08] Gavin King, Pete Muir, Norman Richards, Shane Bryzak, Michael Yuan, Mike Youngstrom, Christian Bauer, Jay Balunas, Dan Allen, Max Rydahl Andersen, Emmanuel Bernard, Nicklas Karlsson, Daniel Roth, Matt Drees, Jacob Orshalick, and Marek Novotny. *Seam - Contextual Components A Framework for Enterprise Java 2.1.1.GA*, 2.1.1ga edition, December 2008. 69, 85, 169, 172, 220

[Knu05] Holger Knublauch. Ramblings on Agile Methodologies and Ontology-Driven Software Development. In *Workshop on Semantic Web Enabled Software Engineering (SWESE), held at* *the 4th International Semantic Web Conference (ISWC 2005)*, Galway, Ireland, November 2005. 62

[Kol07] Dimitrios S. Kolovos. *Editing EMF models with Exeed (EXtended Emf EDitor)*, May 2007. 56

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In Jeff Gray, Juha-Pekka Tolvanen, and Jonathan Sprinkle, editors, *Proceedings of the 6th OOPSLA Workshop on Domain- Specific Modeling (DSM'06)*, volume 37 of *Computer Science and Information System Reports*, Portland, Oregon, USA, October 2006. University of Jyväskylä. 115

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley Interscience, 2008. 12, 100

[Küh06] Thomas Kühne. Matters of (Meta-) Modeling. *Software and Systems Modeling*, 5(4):369–385, December 2006. 6, 82

[Kur07] Ivan Kurtev. Metamodels: Definitions of Structures or Ontological Commitments? In *Workshop on TOWERS of models. Collocated with TOOLS Europe 2007*, 2007. 8, 91

[KW06] Stefan Kühne and Christian Welzel. Metamodellierung am Beispiel der E-Government-Domäne Meldewesen und Eclipse GMF. In Klaus-Peter Fähnrich, Stefan Kühne, Andreas Speck, and Julia Wagner, editors, *Integration betrieblicher Informationssysteme: Problemanalysen und Lösungsansätze des Model-Driven Integration Engineering*, volume IV of *Leipziger Beiträge zur Informatik*, pages 59–72. Eigenverlag Leipziger Informatik-Verbund (LIV), Leipzig, Germany, September 2006. 11

[Lad03] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, July 2003. 102, 103, 107, 116

[LC04] Christian Lange and Michael Chaudron. An Empirical Assessment of Completeness in UML Design. PDF. In *Proceedings of the 8th Conference on Empirical Assessment in Software Engineering (EASE04)*, May 2004. 67

[LG08a] Henrik Lochmann and Birgit Grammel. HybridMDSD - Multi-Domain Engineering with Ontological Foundations. In *The 7th International Semantic Web Conference; Semantic Web Enabled Software Engineering (SWESE 2008)*, 2008. 79

[LG08b] Henrik Lochmann and Birgit Grammel. The Sales Scenario: A Model-Driven Software Product Line. In *Software Engineering 2008 Workshopband*, volume 122, February 2008. 79, 153, 155, 182, 221

[LH01] Carl Lagoze and Jane Hunter. The ABC Ontology and Model. *Journal of Digital Information*, 2(2), 2001. Article No. 77. 81, 89, 220

[LH09] Henrik Lochmann and Anders Hessellund. An Integrated View on Modeling with Multiple Domain-Specific Languages. In *Proceedings of IASTED International Conference on Software Engineerng (SE 2009)*, 2009. 51, 58, 79, 220

[LKL02] Kwanwoo Lee, Kyo C. Kang, and Jaejoon Lee. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In *Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools*, volume 2319 of *LNCS*, pages 62–77, Berlin Heidelberg, January 2002. Springer. 19, 20

[LKT04] Janne Luoma, Steven Kelly, and Juha-Pekka Tolvanen. Defining Domain-Specific Modeling Languages: Collected Experiences. In Juha-Pekka Tolvanen, Jonathan Sprinkle, and Matti Rossi, editors, *Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM04)*, number TR-33 in Computer Science and Information System Reports, Technical Reports, Vancouver, Canada, October 2004. University of Jyväskylä. 111

[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. 11

[LMTS02] Sergio Luján-Mora, Juan Trujillo, and Il-Yeol Song. Extending the UML for Multidimensional Modeling. In *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 290–304, London, UK, 2002. Springer-Verlag. 67

[Loc07] Henrik Lochmann. Towards Connecting Application Parts for Reduced Effort in Feature Implementations. In *Proceedings of 2nd IFIP Central and East European Conference on Software Engineering Techniques (CEE-SET 2007), Posen, Poland*, October 2007. 79, 82, 89

[Ltd09] Sparx Systems Pty Ltd. Enterprise Architect. http://www.sparxsystems.com/products/ea/index.html, 2009. 67

[Luy04] Kris Luyten. *Dynamic User Interface Generation for Mobile and Embedded Systems with Model-Based User Interface Development.* PhD thesis, transnationale Universiteit Limburg, School voor Informatietechnologie, August 2004. 85

[LVM$^+$04] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Víctor López Jaquero. USIXML: a Language Supporting Multi-Path Development of User Interfaces. In *Proceedings of 9th IFIP Working Conference on Engineering for Human-Computer Interaction, jointly with 11th International Workshop on Design, Specification, and Verification of Interactive Systems (EHCI-DSVIS'2004), Hamburg, July 11-13*, volume 3425 of *LNCS*, pages 200–220, Berlin, 2004. Springer. 85

[MBAL07] Zoltan Molnar, Daniel Balasubramanian, and Akos Ledeczi. An Introduction to the Generic Modeling Environment. In *Proceedings of Model-Driven Development Tool Implementers Forum (MDD-TIF07) co-located to TOOLS07*, 2007. 11

[MBB02] Stephen J. Mellor, Marc J. Balcer, and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architecture.* Addison Wesley, 2002. 66

[MBG$^+$03] Claudio Masolo, Stefano Borgo, Aldo Gangemi, Nicola Guarino, and Alessandro Oltramari. WonderWeb - Ontology Infrastructure for the Semantic Web, Deliverable D18: Ontology Library (final). Technical report, Laboratory For Applied Ontology - ISTC-CNR, Trento, Italy, December 2003. 87, 88, 220

[MBH$^+$04] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. *OWL-S: Semantic Markup for Web Services.* World Wide Web Consortium (W3C), November 2004. W3C Member Submission, http://www.w3.org/Submission/OWL-S. 75

[(MD09] Mozilla Developer Center (MDC). XUL - XML User Interface Language. https://developer.mozilla.org/en/XUL, 2009. 85

[MDA03] Object Management Group (OMG). *MDA Guide Version 1.0.1*, June 2003. OMG document omg/2003-06-01, http://www.omg.org/docs/omg/03-06-01.pdf. 7, 13

[Mer07] Ed Merks. Is EMF going to replace MOF? http://ed-merks.blogspot.com/2007/10/is-emf-going-to-replace-mof.html, October 2007. 11

[Met07] MetaCase. Domain-Specific Modeling with MetaEdit+: 10 Times Faster Than UML. White Paper, MetaCase Consulting, 2007. http://www.metacase.com/papers/Domain-specific_modeling_10X_faster_than_UML.pdf. 45

[Met09a] MetaCase. MetaCase: Domain-Specific Modeling with MetaEdit+. http://www.metacase.com/, 2009. 12, 50

[Met09b] MetaCase. *MetaEdit+ Version 4.5 Workbench Users Guide.* MetaCase, 2009. Product Manual, http://www.metacase.com/support/45/manuals/mwb/Mw.html. 12

[MG06] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, March 2006. 9, 223

[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. 15, 82, 117

[Mic08] Sun Microsystems. Java Platform, Enterprise Edition (Java EE), Enterprise JavaBeans

Technology. http://java.sun.com/products/ejb/, 2008. 41

[Mic09a] Sun Microsystems. Java EE at a Glance. http://java.sun.com/javaee/, April 2009. 69

[Mic09b] Sun Microsystems. Java SE at a Glance. http://java.sun.com/javase/, August 2009. 102

[Mic09c] Sun Microsystems. Java Server Pages Technology. http://java.sun.com/products/jsp/, April 2009. 69

[Mic09d] Sun Microsystems. JavaServer Faces Technology. http://java.sun.com/javaee/javaserverfaces/, July 2009. 105

[MM04] Frank Manola and Eric Miller. *RDF Primer*. World Wide Web Consortium (W3C), February 2004. W3C Recommendation, http://www.w3.org/TR/rdf-primer/. 26

[MOF06a] Object Management Group (OMG). *Meta Object Facility (MOF) Core Specification, Version 1.4.1*, January 2006. OMG document formal/05-05-05, http://www.omg.org/docs/formal/05-05-05.pdf. 7, 115

[MOF06b] Object Management Group (OMG). *Meta Object Facility (MOF) Core Specification, Version 2.0*, January 2006. OMG document formal/06-01-01, http://www.omg.org/docs/formal/06-01-01.pdf. 7, 8, 11

[Mol03] Pedro J. Molina. A Review to Model-Based User Interface Development Technology. 2003. 84

[Mor01] Alexandra Weber Morales. Executable Models Are Inevitable. September 2001. 66

[MPS02] Giulio Mori, Fabio Patterno, and Camen Santoro. CTTE: Support for Developing and Analyzing Task Models for Interactive System Design. *IEEE Transactions on Software Engineering*, 28:1–17, 2002. 84

[Mur04] Craig Murphy. Adaptive Project Management Using Scrum. *Methods and Tools*, 12(4):10–22, 2004. 110

[MvH04] Deborah L. McGuinness and Frank van Harmelen. *OWL Web Ontology Language Overview*. World Wide Web Consortium (W3C), February 2004. W3C Recommendation, http://www.w3.org/TR/2004/REC-owl-features-20040210/. 26

[Myl92] John Mylopoulos. Conceptual Modelling and Telos. In Pericles Loucopulos and Roberto Zicari, editors, *Conceptual Modeling, Databases, and Case: An Integrated View of Information Systems Development*, Wiley Professional Computing. John Wiley & Sons Inc, 1992. 83

[Nay07] Keyvan Nayyer. Introduction to Windows Presentation Foundation and XAML. *The Panel: Creating front end experiences*, 2007. 44, 85, 105

[Net07] Code Generation Network. Code Generation Network Interview with Axel Uhl. http://www.codegeneration.net/cg2007/interviews/AxelUhlInterview.pdf, 2007. 50

[NH97] Natalya Fridman Noy and Carole D. Hafner. The State of the Art in Ontology Design: A Survey and Comparative Review. *AI Magazine*, 18(3):53–74, 1997. 24

[NM09] Inc. No Magic. MagicDraw. http://www.magicdraw.com/, 2009. 67

[OB06] Andreas L. Opdahl and Giuseppe Berio. Interoperable language and model management using the UEML approach. In *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management, Shanghai, China*, pages 35–42, New York, NY, USA, 2006. ACM Press. 24, 61

[Obe06] Daniel Oberle. *Semantic Management of Middleware*, volume 1 of *Semantic Web and Beyond: Computing for Human Experience*. Springer, Berlin, 2006. 22

[OCL06] Object Management Group (OMG). *Object Constraint Language, Version 2.0*, May 2006. OMG document formal/06-05-01, http://www.omg.org/docs/formal/06-05-01.pdf. 7, 52

[oD09] Berkeley Institute of Design. prefuse Information Visualization Toolkit. http://prefuse.org/, May 2009. 64, 149

[ODM06] Object Management Group (OMG). *Ontology Definition Metamodel Specification*, October 2006. Final Adopted Specification. OMG document number ptc/06-10-11, http://www.omg.org/docs/ptc/06-10-11.pdf. 7, 75

[OHS02] Andreas L. Opdahl and Brian Henderson-Sellers. Ontological Evaluation of the UML Using the Bunge–Wand–Weber Model. *Software and Systems Modeling*, 1(1):43–67, September 2002. 61, 77, 85, 86

[Old06] Jo Oldovik. *MOFScript User Guide*, November 2006. Version 0.6 (MOFScript v 1.1.11). 10, 13

[OMG] Object Management Group. http://www.omg.org. 7

[OMG01] OMG. *OMG Unified Modeling Language Specification*. Object Management Group, September 2001. 66

[Omo09] Omondo, Inc. OMONDO - The Live UML Company. http://www.eclipsedownload.com/, August 2009. 135

[Ont07] Onto-Med Research Group: Ontologies in Medicine - Foundations, Development and Computer-based Applications. http://www.onto-med.de, 2007. 86

[Opd06] Andreas L. Opdahl. Response to Wyssusek's "On Ontological Foundations of Conceptual Modelling". *Scandinavian Journal of Information Systems*, 18(1):95–106, 2006. 78, 85

[OSG07] OSGI Alliance. http://www.osgi.org, 2007. 11

[ot09] The openArchitectureWare team. openarchitectureware.org. http://www.openarchitectureware.org/, 2009. Official Homepage. 10, 13, 107

[Pal00] William J. Palm. *Introduction to MATLAB 6 for Engineers*. McGraw-Hill Higher Education, 2000. 15

[Par76] David L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2:1–9, March 1976. 19

[PC05] Didier Parigot and Carine Courbis. Domain-Driven Development: the SmartTools Software Factory. Technical Report 5588, Thème COM - Systèmes communicants, June 2005. 16

[PCG$^+$08] Christoph Pohl, Anis Charfi, Wasif Gilani, Steffen Göbel, Birgit Grammel, Henrik Lochmann, Andreas Rummler, and Axel Spriestersbach. Aspect-Oriented Software Development in Business Application Engineering. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development, AOSD (IT)*, 2008. 79, 153

[PDFG92] Dr. Rubén Prieto-Díaz, Dr. Bill Frakes, and Mr. B.K. Gogia. DARE A Domain Analysis and Reuse Environment. Phase i final report, August 1992. 117

[PF02] Sabri Pllana and Thomas Fahringer. On Customizing the UML for Modeling Performance-Oriented Applications. In *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 259–274, London, UK, 2002. Springer-Verlag. 67

[PMM97] F. Paterno, C. Mancini, and S. Meniconi. ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In *Proceedings Interantional Conference on Human-Computer Interaction, Sydney, Australia, 14-18 July*. Chapman & Hall, 1997. 84

[PS08] Eric Prud'hommeaux and Andy Seaborne. *SPARQL Query Language for RDF*. World Wide Web Consortium (W3C), January 2008. W3C Recommendation, http://www.w3.org/TR/rdf-sparql-query/. 52, 62, 136

[PSHH04] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. *OWL Web Ontology Language Semantics and Abstract Syntax*. World Wide Web Consortium (W3C), February 2004. W3C Recommendation, http://www.w3.org/TR/owl-absyn/. 26

[pur06] pure-systems GmbH. Technical white paper: Variant management with pure::variants. 2006. Publicly available under http://www.pure-systems.com/fileadmin/downloads/

pv-whitepaper-en-04.pdf. 129

[pur09] pure-systems GmbH. *pure::variants User's Guide; Version 3.0 for pure::variants 3.0*, 2009. Publicly available under http://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf. 129

[Qua03] Terry Quatrani. Introduction to the Unified Modeling Language. Technical report, IBM, June 2003. 66

[QVT07] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification*, November 2007. Final Adopted Specification ptc/07-07-07, http://www.omg.org/docs/ptc/07-07-07.pdf. 7, 11, 71

[Rac09] Racer Systems. RacerPro OWL reasoner and inference server. http://www.racer-systems.com/, 2009. 52

[RB06] Stephan Roser and Bernhard Bauer. An Approach to Automatically Generated Model Transformations Using Ontology Engineering Space. In *2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006), held at the 5th International Semantic Web Conference (ISWC 2006)*, Athens, GA, USA, November 2006. 72, 75

[RHM09] LLC Red Hat Middleware. *JBoss jBPM 2.0 jPdl Reference Manual*. Red Hat Middleware, LLC, 2009. provided online at http://www.jboss.com/products/jbpm/docs/jpdl/. 69

[Roy70] Winston W. Royce. Managing the development of large software systems. In *Proceedings, IEEE WESCON*, 1970. 110

[RP97] Rational and UML Partners. *UML Version 1.1*, 1997. 52, 66

[Rul09] The Rule Markup Initiative. http://www.ruleml.org, August 2009. 29

[Sea04] Andy Seaborne. *RDQL - A Query Language for RDF*, January 2004. W3C Member Submission, http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/. 62, 136

[SEI08] Carnegie Mellon University Software Engineering Institute. Software Product Lines. http://www.sei.cmu.edu/productlines/index.html, 2008. 19

[SHNM04] Matthew Scarpino, Stephen Holder, Stanford Ng, and Laurent Mihalkovic. *SWT/JFace in Action: GUI Design with Eclipse 3.0 (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004. 166

[SK06] Friedrich Steimann and Thomas Kühne. Coding for the Code: Can models provide the DNA for software development? *ACM Queue*, pages 45–51, December/January 2005-2006. 83

[SKM96] Kelly Steven, Lyytinen Kalle, and Rossi Matti. MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In *CAiSE ;96: Proceedings of the 8th International Conference on Advances Information System Engineering*, pages 1–21, London, UK, 1996. Springer-Verlag. 12

[SMI09] Stanford University School of Medicine Stanford Medical Informatics. The Protégé Ontology Editor and Knowledge Acquisition System. http://protege.stanford.edu, August 2009. 29, 52

[SOEB01] Kirk Schloegel, David Oglesby, Eric Engstrom, and Devesh Bhatt. A New Approach to Capture Multi-model Interactions in Support of Cross-domain Analyses. Technical Report, Honeywell Laboratories, 2001. 72

[Spi00] Diomidis Spinellis. Notable Design Patterns for Domain Specific Languages. *Journal of Systems and Software*, 56(1):91–99, February 2000. 82, 111

[SR08] Jason H. Sharp and Sherry D. Ryan. A Preliminary Conceptual Model for Exploring Global Agile Teams. In *Agile Processes in Software Engineering and Extreme Programming 9th International Conference, XP 2008, Limerick, Ireland, June 10-14, 2008. Proceedings*, pages 137–146, 2008. 110

[Sta08] Leon Starr. How to Build Articulate Class Models and get Real Benefits from UML. *knol - a unit of knowledge.*, August 2008. 67

[Sta09] Leon Starr. Time and Synchronization in Executable UML. *knol - a unit of knowledge.*, March 2009. 67

[SV06] Thomas Stahl and Markus Völter. *Model-Driven Software Development: Technology, Engineering, Management.* Wiley & Sons, 1st edition, 2006. 6, 13, 15

[SVEH07] Thomas Stahl, Markus Völter, Sven Efftinge, and Arno Haase. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management.* dpunkt.verlag, Heidelberg, 2nd edition, 2007. In German. 5, 8, 111, 113, 115, 119, 220

[SWM04] Michael K. Smith, Chris Welty, and Deborah L. McGuinness. *OWL Web Ontology Language Guide.* World Wide Web Consortium (W3C), February 2004. W3C Recommendation, `http://www.w3.org/TR/owl-guide/`. 26, 52, 135, 191

[Sys07] Object Management Group (OMG). *OMG Systems Modeling Language (OMG SysML) Specification, Version 1.0*, March 2007. OMG document ptc/2007-02-03, `http://www.sysml.org/docs/specs/OMGSysML-PAS-07-02-03.pdf`. 83

[TAW03] Peter Tabeling, Rémy Apfelbacher, and Stefan Wappler. *FMC Metamodel The Fundamental Modeling Concepts Metamodel Explained.* FMC Consortium, September 2003. available from `http://www.fmc-modeling.org/download/metamodel/FMC-Metamodel_Explained.pdf`. 83, 155

[TBHLT06] Textuality Tim Bray, Dave Hollander, Andrew Layman, and Richard Tobin. *Namespaces in XML 1.0 (Second Edition).* World Wide Web Consortium (W3C), August 2006. W3C Recommendation, `http://www.w3.org/TR/REC-xml-names/`. 25

[Tea09] TOPCASED Team. TOPCASE, The Open Source Toolkit for Critical Systems. `http://www.topcased.org/`, 2009. 67

[TH09] Dmitry Tsarkov and Ian Horrocks. FaCT++ DL Reasoner. `http://owl.man.ac.uk/factplusplus/`, May 2009. 28, 52

[The09a] The Eclipse Foundation. Eclipse Plug-in Development Environment (PDE). `http://eclipse.org/pde/`, September 2009. 139

[The09b] The Eclipse Foundation. Xtext. `http://www.eclipse.org/Xtext/`, August 2009. 119

[Tol06] Dr. Juha-Pekka Tolvanen. MetaCase: Making Models Work: Domain-Specific Modeling for Full Code Generation. In *Online Proceedings of Model-Driven Development and Product Lines: Synergies and Experience.* University of Leipzig, October 2006. Invited Speaker, `http://software-families.org/slides/tolvanen.pdf`. 12

[UML07] Object Management Group (OMG). *Unified Modeling Language: Superstructure, Version 2.1.1*, February 2007. OMG document formal/2007-02-05, `http://www.omg.org/docs/formal/07-11-02.pdf`. 8, 35, 66, 84, 220

[VG07] Markus Voelter and Iris Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *Proceedings Software Product Line Conference, 2007. SPLC 2007. 11th International*, 2007. 22, 39

[Vis08] Eelco Visser. WebDSL: A case study in domain-specific language engineering. In R. Lammel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Lecture Notes in Computer Science. Springer, 2008. 111, 117, 119

[W3C07] W3C. Semantic Web Activity. `http://www.w3.org/2001/sw/`, 2007. 52

[Wei95] D.M. Weiss. Software Synthesis: The FAST Process. In *In Proceedings of the International Conference on Computing in High Energy Physics, Rio de Janeiro*, 1995. 117

[WGL06] Gerd Wagner, Adrian Giurca, and Sergey Lukichev. A Usable Interchange Format for Rich Syntax Rules Integrating OCL, RuleML and SWRL. In *Reasoning on the Web (RoW2006), Workshop at WWW2006, Edinburgh, UK*, May 2006. 29

[WH08] Jonathon Wong and Rupert Howell. Minilang and OFBiz. *Packt Publishing - Community Experience Distilled*, November 2008. 70

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language – Getting your models ready for MDA*. Object Technology Series. Addison-Wesley Longman, Amsterdam, The Netherlands, 2nd edition, 2003. 7

[WK05] Boris Wyssusek and Helmut Klaus. On the Foundation of the Ontological Foundation of Conceptual Modeling Grammars: The Construction of the Bunge–Wand–Weber Ontology. In *Proceedings of 1st International Workshop on Philosophical Foundations of Information Systems Engineering (PHISE'05), in conjunction with the 17th Conference on Advanced Information System Engineering (CAiSE'05)*, Porto, Portugal, June 2005. 78, 85

[WK06] Jos Warmer and Anneke Kleppe. Building a Flexible Software Factory Using Partial Domain Specific Models. In Jeff Gray, Juha-Pekka Tolvanen, and Jonathan Sprinkle, editors, *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06), Portland, Oregon, USA*, volume 37 of *Computer Science and Information System Reports*, pages 15–22. University of Jyväskylä, October 2006. 15, 16

[Won04] WonderWeb - Ontology Infrastructure for the Semantic Web. `http://wonderweb.semanticweb.org`, July 2004. 87

[WSGT09] Jules White, Douglas C. Schmidt, Aniruddha Gokhale, and The Distributed Object Computing (DOC) group at Vanderbilt University. GEMS Home page. http://www.eclipse.org/gmt/gems/, 2009. 12

[WSNW06] Jules White, Douglas C. Schmidt, Andrey Nechypurenko, and Egon Wuchner. Domain-Specific Intelligence Frameworks for Assisting Modelers in Combinatorically Challenging Domains. In *Proceedings of the Workshop on Generative Programming and Component Engineering for QoS Provisioning in Distributed Systems, workshop at GPCE'06, Portland, Oregon, USA*, October 2006. 12, 52

[WSNW07] Jules White, Douglas C. Schmidt, Andrey Nechypurenko, and Egon Wuchner. Introduction to the Generic Eclipse Modeling System: Developing a Graphical Modeling Tool for Eclipse. *Eclipse Magazine*, 6:11–19, January 2007. 12, 83

[WV07] Xiaofeng Wang and Richard Vidgen. Order and chaos in software development: A comparison of two software development teams in a major company. In *Proceedings of the 15th ECIS*, June 2007. 110

[WW88] Yair Wand and Ron Weber. An ontological analysis of some fundamental information system concepts. In J. I. DeGross and M. H. Olson, editors, *Proceedings of the 9th International Conference on Information Systems*, pages 213–224, Minneapolis, Minnesota, USA, 1988. 77, 85

[WW95] Yair Wand and Ron Weber. On the deep structure of information systems. *Information Systems Journal*, 5(3):203–223, 1995. 85, 86

[Wys06] Boris Wyssusek. On Ontological Foundations of Conceptual Modelling. *Scandinavian Journal of Information Systems*, 18(1), 2006. 85

[XPD05] The Workflow Management Coalition. *XML Process Definition Language, Version 2.0*, October 2005. `http://www.wfmc.org/standards/docs/TC-1025_xpdl_2_2005-10-03.pdf`. 84

[YH07] Michael J. Yuan and Thomas Heute. *JBoss Seam: Simplicity and Power Beyond Java EE*. JBoss Series. Prentice Hall International, May 2007. 16, 69

[You09] Charles Young. Oslo and the OMG. *geekswithblogs.net*, January 2009. 12, 50

[Yua07] Heng Yuan. CookXml. `http://cookxml.yuanheng.org`, 2007. 85

[Zab08] Manuel Zabelt. Konzeption eines konnektorgestützten Software-Entwicklungsprozesses auf Basis mehrerer DSLs. Diplomarbeit, Technische Universität Dresden, Department of Computer Science, March 2008. 79