DISSERTATION / DOCTORAL THESIS

Titel der Dissertation / Title of the Doctoral Thesis

Metamodel Composition in Hybrid Modelling
- A Modular Approach

verfasst von / submitted by

Mag. Srđan Živković

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Doktor der technischen Wissenschaften (Dr. techn.)

Wien, 2016 / Vienna, 2016

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 786 175
Dissertationsgebiet lt. Studienblatt / field of study as it appears on the student record sheet:	Wirtschaftsinformatik
Betreut von / Supervisor:	o. Univ.-Prof. Dr. Dimitris Karagiannis

Abstract

It has been recognised that model-based engineering (MBE) approaches are beneficial for the engineering of increasingly complex systems and software. MBE encourages the usage of modelling languages and methods to analyse, design and develop various kinds of enterprise and software systems. A multitude of standard and domain-specific modelling languages (DSML) are being engineered to facilitate modelling. While focusing on a narrow domain, DSMLs are composed into hybrid modelling languages, to benefit from their combined use, i.e. to address the system complexity from different perspectives holistically. Furthermore, modelling languages in general evolve over time, requiring adaptations, extensions and customisation to accommodate the changing requirements of a problem domain. Language definition is a complex engineering task. Metamodelling has been recognised as a practical yet rigorous formalism for language definition with a metamodel being its pivotal engineering artefact. While current metamodelling techniques provide powerful concepts for creating metamodels from scratch, they lack concepts for more efficient metamodelling. To answer the challenges posed by an increasing need for new modelling languages in general and by their hybrid and evolving nature in particular, this thesis introduces an approach to modular metamodel engineering (MME). Based on the hypothesis that modularisation and composition reduce effort and improve efficiency in metamodel definition, the thesis extends existing metamodelling concepts towards modular definition of metamodels. In particular, the thesis introduces concepts for defining reusable metamodel fragments and a comprehensive set of metamodel composition operators for fragment combination. By promoting reuse, such a modular approach allows for systematic, flexible and efficient definition and customisation of metamodels. The introduced concepts are formalised in a language for modular metamodel engineering (MMEL). A possible realisation of MMEL is explicated based on the ADOxx metamodelling platform. Finally, the MME approach and in particular the language MMEL is evaluated in two case studies of enterprise modelling methods in the context of OMILab. On the one side, MMEL is applied to modularise the hybrid modelling method BPMS. On the other side, MMEL is used to design a hybrid DSML that combines a language for network devices modelling with the ontology language OWL.

Zusammenfassung

Das Modell-basierte Engineering (MBE) stellt einen praktischen Ansatz zur Entwicklung von immer komplexer werdenden Systemen und Software dar. Im Vordergrund des MBE stehen Modellierungssprachen und Methoden, welche für die Analyse, Design und Entwicklung von Enterprise- und Software-Systemen unterschiedlicher Art genutzt werden. Eine Vielzahl von standard- sowie domänenspezifischen Modellierungssprachen werden entwickelt, um die Modellierung zu ermöglichen. Häufig werden solche Sprachen in sogenannte hybride Modellierungsprachen zusammengelegt, um von ihren kombinierten Nutzung zu profitieren. Ferner unterliegen Modellierungssprachen kontinuierlicher Anpassung und Weiterentwicklung, um den ändernden Anforderungen der Problemdomäne stand zu halten. Die Sprachdefinition ist dabei im Grunde eine komplexe Aufgabe. Es ist bekannt, dass die Metamodellierung einen praktischen und zugleich einen rigorosen Formalismus zur Sprachdefinition darstellt, in welchem das Metamodell als Drehpunkt der Spezifikation fungiert. Während die existierenden Metamodellierungsansätze ausgereifte Konzepte zur Metamodelldefinition "vom Grund auf" anbieten, fehlen weitere Konzepte, welche eine effizientere Metamodellierung ermöglichen. Die vorliegende Arbeit sucht diese Lücke mit einem Ansatz zum modularen Metamodell-Engineering (MME) zu schließen. Basierend auf der Hypothese, dass modulare und kompositionale Ansätze den Aufwand minimieren und zugleich die Effizienz und die Flexibilität der Metamodelldefinition steigern, erweitert diese Arbeit bestehende Metamodellierungsansätze mit Konzepten zur modularen Metamodelldefinition. Insbesondere werden Konzepte zur Definition von wiederverwendbaren Metamodell-Fragmenten sowie eine Reihe von umfassenden Operatoren zur Metamodellkomposition eingeführt. Durch wesentlich erhöhte Wiederverwendbarkeit, ermöglicht der modulare Ansatz eine systematische und effiziente Entwicklung und Anpassung von Metamodellen. Die eingeführten Konzepte werden in einer Sprache zum Modularen Metamodell-Engineering (MMEL) formalisiert. Die Realisierbarkeit der MMEL-Sprache wird anhand der ADOxx Metamodellierungsplattform erläutert. Darüber hinaus konnte der MME-Ansatz bzw. die MMEL-Sprache in zwei Fallstudien zur modularen Entwicklung von hybriden Modellierungssprachen im Kontext von OMILab positiv evaluiert werden.

Acknowledgements

"The Loneliness of the Long Distance Runner"

Alan Sillitoe

Writing a dissertation is like running a long-distance run. A long-distance run requires a lot of preparation, but most of all, it requires a great amount of endurance, determination and patience, in order to reach the finish line. On that path, there are many ups and downs, the moments of flow, but also tough moments of weakness and loneliness. Luckily, one is not completely alone on this journey. Your coach supports you in giving you helpful advice, fellow runners share running tips that will support and encourage you along the way, and, finally, you have the audience that somehow always happens to be there when most needed to cheer you up to move on. I see the coach as the thesis supervisor, the runners as your colleagues, and the audience to be your friends and family. Herewith, I would like to express my gratitude to all of them for helping me reaching the finish line of my long-distance run.

First and foremost, I would like to thank my supervisor, Prof. Dr. Dimitris Karagiannis, for introducing me to the field of metamodelling and giving me the opportunity to undertake a research on a very interesting and challenging topic of metamodel composition. I appreciate all the scientific advice that helped me to streamline my research towards its successful completion. I would like to thank Dr. Harald Kühn who gave me the opportunity to work on the metamodelling topics in the industrial setting and who introduced me to the practical challenges of metamodelling. His earlier scientific work on method integration sparked my research interest for metamodel composition that has lasted for years. I thank Prof. Dr. Wilfried Grossmann and Prof. Dr. Hans-Georg Fill, for their time and effort spent evaluating my dissertation and their insightful comments provided.

My thanks go to my PhD colleagues Wilfrid Utz and Christoph Moser, with whom I worked on various PhD assignments at the Department for Knowledge Engineering, Faculty of Computer Science, University of Vienna. I thank my colleagues from BOC Information Systems GmbH, Vienna, with

whom I intensively worked on the ADOxx metamodelling platform all these years. Here, I would like to mention Lukas Ramach with whom I shared numerous discussions on ADOxx concepts. Further thanks go to my undergraduate colleague and friend Marija Bjeković from LIST, and Christos Lekaditis from BOC for their interest and time to comment and proofread the thesis. Furthermore, I would like to thank my former colleagues from the EU research project MOST, with whom I worked on the integrated modelling languages and the application of (meta)modelling in software development. I would like to mention Krzysztof Miksa from Comarch with whom I worked on the hybrid modelling language PDDSL and Christian Wende with whom a shared the passion for language composition and metamodelling tool development.

Finally, I want to thank my friends, relatives and my family for the support and understanding of the time I did not spend with them, for missed moments and celebrations while working an extra hour on the thesis. In particular, I want to thank my mother and my late father who taught me to endure in order to reach the higher goals and my brother for the overall encouragement and support during my entire scholar path. Lastly, my deepest gratitude goes to my wife Christiane and our daughter Helen, for their endless patience and faithful support throughout.

Srđan Živković
Vienna, December 2015

Contents

Part I

Overview

Chapter 1

Introduction

"If we knew what it was we were doing, it wouldn't be called 'research', would it?"

Albert Einstein

1.1 Overview

With the rise of digital computing, which has been the major vehicle of the third industrial revolution started in the late fifties [Wikipedia, 2016], the IT industry today has become the key provider for business growth, driving business productivity and establishing numerous new business fields and models [McDavid, 2004]. Today, the software industry more than ever faces the challenges of providing quality-focused, adaptable, reliable and durable products and services which sustainably align business and IT. Software and system engineering methods, techniques and tools which raise development productivity by leveraging reusability and programming abstraction level towards automation are key factors to achieve a competitive advantage for software providers.

Model-based approaches for system and software engineering deal with the increased system complexity by raising the level of abstraction to models. Modelling may be used descriptively on the level of system analysis and design to better understand the system under consideration and improve its performance. On the other side, modelling is also used prescriptively to interpret or generate executable systems, i.e. program code, with an ultimate goal to automate the software production. Formalisation of concepts needed to model a system under consideration on any abstraction level is always captured in a modelling language. Up until recently, modelling language definition has been reserved for a rather small number of language engineering experts. While the demand for new modelling languages was rather low,

the complexity of creating new languages was high. This state of affairs changed significantly with the rise of domain-specific modelling languages (DSML) and appropriate tools for language engineering. Instead of sticking to a handful of general-purpose one-size-fits-all modelling languages, a plethora of languages with precise semantics specific to a problem domain are being designed in order to facilitate modelling, and respectively model-based analysis and design, and development of systems and software. Language engineering tools such as metamodelling platforms, language workbenches, software factories provide necessary means to facilitate language design and derivation of modelling tools. Hence, (modelling) language engineering, a discipline that deals with systematic design and implementation of languages, gained on significant importance in both research and industry as a vehicle to advance model-based engineering of systems and software to another level.

This thesis represents an attempt to contribute to the general field of modelling method and language engineering, and in particular, to the field of metamodelling (in the following, we refer to it as metamodel-based modelling language engineering (MLE)). Since the central design artefact in MLE is a metamodel, that represents the grammar/abstract syntax of a modelling language [Selic, 2011], the thesis specifically tackles the areas of metamodel modularisation and metamodel composition in modelling language engineering. By allowing for systematic modularisation of metamodels into reusable constructs and their combination via composition, the approach to *Modular Metamodel Engineering (MME)* contributes to the increased reuse of language design artefacts and, in general, to higher efficiency in metamodelling.

1.2 Problem Statement

1.2.1 Hybrid Languages

Demand for DSMLs and corresponding tools is continuously increasing, as more stake-holders seek to leverage the power of models to abstract from boiler plate code and excel sheets when describing and engineering complex systems. Since DSMLs focus on and are restricted to a narrow single domain, the demand to use them in combination with other languages becomes obvious. We call such combined, integrated languages *hybrid languages*. For example, in various enterprise process modelling and architecture frameworks (Zachman Framework [Zachman, 1987], TOGAF [The Open Group, 2012]) different system aspects are described using a set of interrelated concepts that are combined together to provide a holistic view of the system. In [Visser, 2008] a set of textual DSLs is built to capture different aspects of web applications, which when combined together, represent a complete specification to generate a target web application system. Even the languages

stemming from different technical spaces may be combined, to benefit from each others strengths. In [Zivkovic et al., 2011, Zivkovic et al., 2015] a DSML for managing network devices is combined with ontology language OWL to provide better specification of domain constraints. Hence, to allow for definition of hybrid languages, systematic modularisation concepts for metamodels are required in order to handle DSMLs or their fragments as reusable, recombinable language engineering units. Furthermore, comprehensive, flexible composition concepts are missing that would allow for hybrid combination of such DSML metamodels.

1.2.2 Evolving Languages

Besides the hybrid characteristic, (modelling) languages, in general, undergo changes during their life cycle, i.e. they evolve. Such *evolving languages* may be *adapted*, *customised* to some problem and project-specific needs, or may be *extended* and improved with additional features and constructs. Standard languages such as BPMN [OMG, 2013] for business process modelling, ArchiMate [The Open Group, 2013] for enterprise architecture modelling, UML [OMG, 2011a,OMG, 2011b] for software modelling, MOF [OMG, 2014] for language modelling, OWL [Motik et al., 2009b] for formal ontology modelling, or the proprietary, industry-strength languages for enterprise modelling such as BPMS [BOC, 2015], have been evolving over time introducing new versions to adopt various kinds of extensions and improvements. Furthermore, released language versions are further customised to suit problem and project-specific needs. For example, a company may adopt BPMN 2.0 [OMG, 2013] as a standard for business process modeling but requires company-specific extensions for process-based risk management. Such customisation may involve introduction of additional risk-related properties to existing language entities, creation of new entities or even integration with proprietary languages to build a custom hybrid solution. Ideally, such custom extensions should be portable to the upcoming version of the base language. One of the challenges when developing evolving languages over time is the ability to *systematically modularise* such languages and define language core parts, extension points and extensions/customisations, to modularise such language blocks to be reused over different language versions and to allow for flexible substitution and combination of such language modules [Živković and Karagiannis, 2016].

1.2.3 Metamodelling-in-the-Large vs. Metamodelling-in-the-Small

Metamodel-based language engineering, i.e. metamodelling faces challenges that arise from the nature of hybrid and evolving languages. While language definition is in its nature a complex engineering task, the demand for hy-

brid and evolving languages adds to the complexity even further. Language definitions as complex monolithic design artefacts are hard to comprehend, maintain, adapt and integrate. To quote Jackson [Jackson, 1990], "having divided to conquer, we must reunite to rule", a common way to deal with system complexity is to break down the system into smaller, ideally, modular pieces that can be combined together to build a complex system. Focusing on systems built out of interacting components instead of monolithic ones, is essential to the engineering of complex systems. The necessity of introducing component-oriented concepts in programming languages has been nicely pointed out in [DeRemer and Kron, 1976]. They claimed that while having programming constructs for creating small (monolithic) programs and modules is essential (programming-in-the-small), it is equally important to have constructs for combining single modules into an integrated whole (programming-in-the-large) in order to build large and complex software. If we apply the metaphor of programming-in-the-small vs. programming-in-the-large to the field of metamodelling, it may be stated that current metamodelling languages provide primarily concepts for "metamodelling-in-the-small". While the major focus is set on providing the core constructs for metamodelling, less support is offered for the modular and compositional engineering of metamodels [Živković and Karagiannis, 2015]. Although standard metamodelling languages such as MOF recognised these issues and introduced basic modularisation concepts such as package and composition operators such as package import and package merge, comprehensive concepts for the definition of component-like metamodel modules and corresponding flexible composition operators for language composition are still missing. It has been recognised that modularisation and component-oriented, compositional development of software systems [Szyperski, 2002, Aßmann, 2003] addressed the issues of programming-in-the-small and the pushed software industry into a new era of software factories [Greenfield et al., 2004] and component-based software assembly. Hence, approaches that would enable component orientation in metamodelling are sought, in order to contribute to the idea of "metamodelling in-the-large".

1.2.4 Research Questions

The aim of the underlying work is to extend metamodelling approaches with modular principles in order to allow for flexible and efficient construction of complex, hybrid and evolving metamodels. Therefore, *the main research question is whether and how the metamodelling languages can be extended to support modular, compositional engineering of metamodels?* We refer to the definition of software composition systems introduced by [Aßmann, 2003] to postulate some concrete research questions. There, a modular, composition system is defined as a triple of a component model, a composition language, and a composition technique. Whereas the component model de-

scribes modules that are to be composed, composition language provides constructs to specify the composition of modules. Composition technique deals with mechanisms needed to perform the composition, i.e. to derive the output of the composition, a composite module. Following these tree modularisation and composition elements, the following research questions are postulated:

- *RQ1: Which concepts are needed to introduce modularisation in metamodel engineering and how?* While current languages for metamodel definition, i.e. metamodelling languages such as MOF [OMG, 2014], GOPPRR [Kelly et al., 1996], ADOxx Meta^2-Model [Kühn, 2010, ADOxx, 2015], GME [Ledeczi et al., 2001a], EMF [Steinberg et al., 2008] or GrUML [Ebert et al., 1996] provide basic constructs to define white-box metamodel packages, advanced concepts for the definition of black-box metamodel fragments with well-defined interfaces are missing [Živković and Karagiannis, 2015].

- *RQ2: Which concepts are needed to introduce flexible composition in metamodel engineering?* Many of the current metamodelling languages introduced before support inheritance or inheritance-like composition operators when composing parts of metamodels. The deficiencies of the inheritance (single or multiple) when used in context of component composition such as the violation of information hiding and construction of fragile, hard-wired dependencies, have been extensively discussed in [Szyperski, 2002]. Metamodel design that relies solely on complex inheritance hierarchies is fragile and suffers from the "house-of-card architecture"[1] syndrom. Besides that, although the reuse of structural features by subclassing is one of the main advantages of inheritance, it may at the same time be its major drawback, too. Introduction of derived types (subclasses) for the purpose of extending the base class with additional features may lead to complex class hierarchies and over-engineered metamodels. Furthermore, extending a class by subclassing may in some cases not be possible (single inheritance restriction, or "sealed" base classes) or not desired (the base class is already in use, i.e. instances exist that would require tool recompilation and model migration). Furthermore, multiple inheritance has been discussed controversially since its introduction in programming languages [Bracha, 1992], as well as, more recently, in metamodelling [Selic, 2011]. Multiple inheritance is criticised for an increased unanticipated complexity and ambiguity in class design, allowing for anti-patterns such as "diamond inheritance problem" and

[1] According to Wikipedia [Wikipedia, 2015], the House-of-card architecture is a fragile structure... a structure on a shaky foundation or one that will collapse if a necessary (but possibly overlooked or unappreciated) element is removed.

over-generalisation. Finally, other standard operators such as aggregation, import and merge exist, however, all of them operate on white-box components, which lead to hard-wiring of metamodels. Hence, to mitigate the above mentioned issues, composition operators that allow for more flexible composition beyond inheritance and pure white-box operators are needed [Živković and Karagiannis, 2016].

- *RQ3: How can metamodel composition be realised within metamodelling platforms?* Environments for metamodel-based language definition, i.e. metamodelling platforms, such as ADOxx [ADOxx, 2015], Eclipse [Eclipse, 2013, Gronback, 2009], GME [Ledeczi et al., 2001a] or MetaEdit+ [Kelly et al., 1996, MetaCase, 2011] provide core mechanisms for metamodel and language definition. Mechanisms that allow for metamodel composition, i.e. derivation of composite metamodels need to be seamlessly integrated.

1.3 Thesis Contributions

The underlying thesis answers the previously formulated research questions by introducing an approach to *modular metamodel engineering (MME)*. The main contribution of the thesis, the MME approach, consists of a *concept*, a *language* and a *realisation* within the metamodelling platform ADOxx.

1.3.1 A Concept for Modular Metamodel Engineering

The concept for *modular metamodel engineering (MME)* is a continuation of the fragment-based method integration idea proposed by Kühn [Kühn, 2004]. Focusing on the language part of the method, and in particular, on the metamodel as a pivotal element in language definition, MME allows for modular definition of metamodels. MME aims at extending the metamodelling concepts, i.e. the constructs of a metamodelling language, with constructs for the modular metamodel definition. On the one side, it introduces concepts to systematically define reusable, self-contained metamodel fragments. On the other side, it extends metamodelling languages with a set of composition operators to holistically support white-box, grey-box and black-box composition. Finally, it follows the purely interpretative composite language derivation as a chosen composition technique to realise both invasive and non-invasive metamodel composition.

1.3.2 A Language for Modular Metamodel Engineering

The concepts of MME are formalised in a language for modular metamodel engineering (MMEL), which represents an extension of metamodelling languages towards modular orientation. It consists of two major sublanguages,

a metamodel modularisation language, and a metamodel composition language. The *metamodel modularisation language* introduces concepts for encapsulating parts of metamodels into so called *metamodel fragments* with *well-defined interfaces* and *explicit dependencies*, in order to realise the idea of self-contained, prefabricated, black-box metamodel components. On the other side, the *metamodel composition language* introduces a composition algebra, i.e. a comprehensive set of operators for the flexible composition of metamodel fragments. The composition language is holistic as it consists of sublanguages for diverse composition scenarios such as black-box, grey-box, and white-box metamodel composition. *Black-box composition* operators allow for the composition of metamodel fragments on the level of explicit interfaces based on the ideas of the component-oriented software development. In addition, based on the notion of invasive software composition [Aßmann, 2003], *grey-box composition operators* are introduced for the composition of metamodel fragments based on implicit interfaces. Grey-box composition operators complement black-box operators to allow for flexible fragment combination in the cases where explicit interfaces are not planned or do not exist. Finally, advanced *white-box composition operators* are introduced to flexibly compose internals of fragments or for composition scenarios where other composition operators seem not to be sufficient.

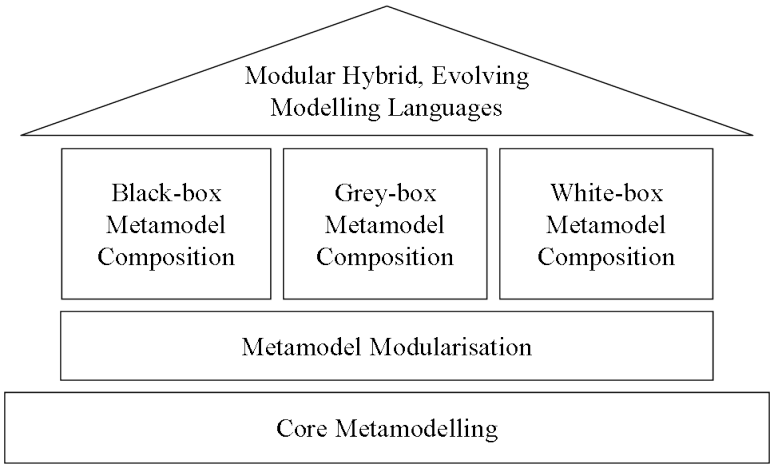


Figure 1.1: Metamodel composition for modular engineering of hybrid, evolving modelling languages

Figure 1.1 illustrates the anatomy of the approach and its main contributions. While *core metamodelling* refers to the basic metamodelling concepts for constructing metamodels, i.e. for "metamodelling-in-the-small", *metamodel modularisation* contributes to that core with concepts for encapsulation and information hiding in metamodelling. On top of modularisation concepts, comprehensive *metamodel composition* operators for black-box, grey-box and white-box composition allow for flexible combination of meta-

models and metamodel fragments to support the idea of "metamodelling-in-the-large" and to allow for the construction of modular (hybrid, evolving) modelling languages.

1.3.3 Realisation in Metamodelling Platforms

Furthermore, the introduced language for modular engineering of metamodels in general may only be useful if it seamlessly integrates into the existing mechanisms for metamodel definition of the metamodelling platforms. Based on the metamodelling platform ADOxx, we explicate how a metamodelling platform can be extended to support metamodel modularisation and composition natively, such that no external mechanisms nor generative techniques are needed to derive composite metamodels. Even though we focus only on one platform, the approach and the language itself are generic such that the idea may be applied to other metamodelling and language engineering environments, too.

Furthermore, we discuss how metamodel composition fills in the missing piece of the puzzle for the realisation of the more general concept of Metamodel and Functionality Building Blocks (MFB) in ADOxx.

1.3.4 Evaluations

The applicability of the modular approach for metamodel engineering in general and of the introduced language MMEL in particular is demonstrated in two case studies in the context of OMILab (Open Model Initiative Laboratory) [OMI, 2015]. The selection of modelling languages for systematic evaluation in both case studies reflects the major duality of modelling concerns (structure, behaviour) and the diversity of modelling domains in enterprise modelling (Business Processes (BP), Enterprise Architecture (EA)).

- *Evaluation I: Case Study in BP: Modular BPMS.* The Business Process Modelling Systems (BPMS) method is an integrated, hybrid modelling method used for enterprise-wide business modelling of four core enterprise areas business processes, products, organisations and information technology [Karagiannis, 1995,Bayer and Kühn, 2013,BOC, 2015]. We demonstrate the applicability of the introduced approach by modularising the BPMS. By doing so, we focus on the BPMN [OMG, 2013] fragment of the BPMS and show how it is embedded into BPMS [Zivkovic et al., 2007, Rausch et al., 2011], this time using the introduced modularisation and metamodel composition concepts.

- *Evaluation II: Case Study in EA: Hybrid PDDSL.* One of the results of the EU research project MOST has been a technology for the integration of ontology and modelling languages. Combined with DSMLs, ontology languages provide formal and expressive means to describe

the particularities of the problem domain [Walter and Ebert, 2009]. In one of the case studies, a concrete DSML, the PDDSL, for modelling and configuration of network devices is extended by the ontology language OWL2 [Motik et al., 2009b] to allow for and ease the semantic constraint definition of valid device configurations [Zivkovic et al., 2011, Miksa et al., 2013, Zivkovic et al., 2015]. We revisit this case study and demonstrate the usefulness of the modularisation and metamodel composition by constructing the s.c. Hybrid PDDSL in a modular way.

1.4 Thesis Structure and Additional Information

In the following, we provide an overview of the thesis structure, together with some advises on how to read it. Furthermore, we define what is the scope of the underlying work and what is not. Finally, a list of publications related to the thesis is given.

1.4.1 Thesis Structure

Following this introductory part (*Part I: Overview*), the thesis is divided into three main parts (see Fig. 1.2).

Part II: Existing Work

Part II provides an overview of existing concepts that represent the background of the underlying work.

Chapter II introduces the basic concepts related to the existing work on the engineering of modelling methods. After introducing the basic notions of a modelling method, the focus is set on the method integration approaches for hybrid modelling methods.

Chapter III elaborates on the modelling language definition, the formalism which is, in this work, subject to extension by the modularisation and compositional aspects. The chapter introduces the basic notions of a modelling language. In doing so, it particularily positions the metamodel as a pivotal element in language definition. After the basic notions are introduced, an overview and a comparative analysis of metamodelling languages, i.e. of languages for metamodel definition is given.

Chapter IV is devoted to metamodelling environments as a crucial realisation technology for modelling language definition, and in particular, for (modular) metamodel definition. After the discussion on the particularities of such environments, a comparative analysis of a selection of metamodelling platforms is provided based on a classification framework.

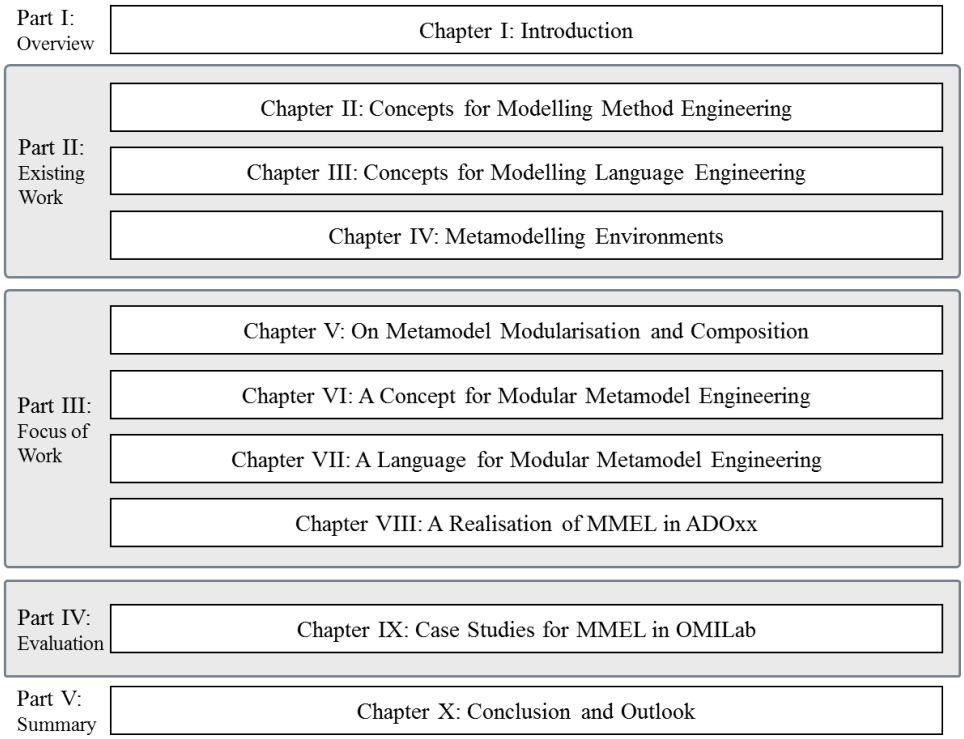


Figure 1.2: Thesis structure

Part III: Focus of Work

Part III represents the main contribution of the thesis and is divided into four chapters as follows.

Chapter V provides basic concepts and an overview of the existing work on metamodel modularisation and composition in the context of metamodel-based modelling language engineering. After the basic elements of modular systems are covered, we elaborate on various composition operators and their particularities, as found in the literature. Finally, based on an introduced classification framework, we analyse related work on metamodel modularisation and composition.

In *Chapter VI* a concept for modular metamodel engineering is introduced. After the key requirements and notions of a modular approach for metamodel definition are introduced, we elaborate on the metamodel modularisation and metamodel composition concepts, as fundamental aspects of the approach. We introduce the notion of a metamodel fragment having explicitly defined interfaces and explicit dependencies only, and define metamodel composition operators that work on such fragments to allow for their combination.

Chapter VII formalises the ideas introduced in the previous chapter in a

language for modular metamodel engineering, which represents an extension to metamodelling languages and the cornerstone of the introduced approach. After sketching the general architecture of the language, we specify the language by defining its congruent elements (syntax, semantics and notation). The language itself consists of sublanguages for metamodel modularisation and metamodel composition.

Chapter VIII elaborates on a realisation of the language for modular metamodel engineering as introduced in the previous chapter. The ADOxx metamodelling platform is used as the underlying environment for realisation. In particular, it is specified how the ADOxx metamodelling language (aka ADOxx Meta^2-Model) is extended by the modularisation and composition metamodelling capabilities, in order to allow for the native, interpretative derivation of composite metamodels.

Part IV: Evaluation

Part IV elaborates on the evaluation of the modular metamodel engineering approach. Hence, in *Chapter IX*, we introduce two evaluation case studies and discuss the benefits of using modular metamodel engineering.

Finally, in *Chapter X* (*Part V: Summary*) we summarise the work and provide outlook on future work.

1.4.2 Scope: Additional Comment

In the following, additional comments on the scope of the underlying work are given.

- *Contribution to metamodelling.* It is important to note that the underlying work is not about a specific hybrid, composite modelling language or metamodel. Instead, the focus of the work is on providing concepts for defining arbitrary hybrid metamodels using systematic modularisation and metamodel composition. That said, it represents the extension of core metamodelling, i.e. of a metamodelling language (language to define metamodels). Hence, the concepts introduced are concepts on the M3 level of the four-layered architecture of model-based approaches.

- *Focus on metamodels.* Although, in this work, we interchangeably talk about modelling methods, modelling languages and metamodels, the primary focus is on metamodel definition. However, while metamodel represents a core, pivotal artefact in language definition and consequently for modelling method definition, the modular concepts of metamodels influence indirectly the modularity of modelling languages and methods. In other words, modular metamodel engineering is a prerequisite for modular language and method engineering.

- *Realisation in metamodelling platforms.* While the approach, in particular, the language for modular metamodel engineering may be applied on various metamodelling platforms, we focus in this work particularly on the ADOxx metamodelling platform. Nevertheless, in our attempt to define a common core metamodelling language that abstracts from specific metamodelling implementations, and that is used as the base for the extensions, we addressed the most common and some variable features of those metamodelling platforms and languages, such that, with some restrictions or extensions, the modular idea as introduced in this work can, indeed, be instantiated within other platforms, as well.

1.4.3 Publications

This thesis has been created based on the authors research and practice in the field of metamodelling and metamodelling-based modelling tool development. In the following, the authored and co-authored peer-reviewed publications are listed related to the topics addressed in the thesis.

On modularisation and composition of metamodels:

- Živković, S., and Karagiannis, D. (2016). *Mixins and Extenders for Modular Metamodel Customisation.* Accepted for ICEIS 2016, Rome, Italy, 2016.
- Živković, S., and Karagiannis, D. (2015). *Towards Metamodelling-in-the-Large: Interface-based Composition for Modular Metamodel Development.* In Enterprise, Business-Process and Information Systems Modeling - 20th International Conference, EMMSAD 2015, Held at CAiSE 2015, Stockholm, Sweden, June 8-9, 2015, Proceedings, (pp. 413-428), Springer International Publishing.
- Zivkovic, S., Kühn, H., and Karagiannis, D. (2007). *Facilitate Modelling Using Method Integration: An Approach Using Mappings and Integration Rules.* In üsterle, H., Schelp, J., and Winter, R., editors, European Conference on Information Systems, ECIS2007. University of St. Gallen, Switzerland.

On hybrid modelling languages:

- Zivkovic, S., Miksa, K., and Kühn, H. (2015). *On Developing Hybrid Modeling Methods using Metamodeling Platforms: A Case of Physical Devices DSML Based on ADOxx.* International Journal of Information System Modeling and Design (IJISMD), 6(1), 47-66. IGI Global, Hershey, PA, USA.
- Zivkovic, S., Miksa, K., and Kühn, H. (2011). *A Modelling Method for Consistent Physical Devices Management: An ADOxx Case Study.*

In Salinesi, C. and Pastor, O., editors, Advanced Information Systems Engineering Workshops - CAiSE 2011 International Workshops, London, UK, June 20-24, 2011. Proceedings, volume 83 of Lecture Notes in Business Information Processing, pages 104-118. Springer.

- Kühn, H., Murzek, M., Specht, G., and Zivkovic, S. (2011). *Model-driven Development of Interoperable, Inter-organisational Business Processes.* In Charalabidis, Y., editor, Interoperability in Digital Public Services and Administration: Bridging E-Government and E- Business, pages 119-143. Hershey, PA, USA.

On modularisation in metamodelling platforms:

- Wende, C., Aßmann, U., Zivkovic, S., and Kühn, H.(2011). *Feature-based Customisation of Tool Environments for Model-driven Software Development.* In de Almeida, E. S., Kishi, T., Schwanninger, C., John, I., and Schmid, K., editors, Proceedings of Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011, pages 45-54. IEEE.

On metamodelling platforms technology:

- Zivkovic, S., Wende, C., Thomas, E., Parreiras, F., Walter, T., Miksa, K., Kühn, H., Schwarz, H., and Pan, J. (2013). *A Platform for ODSD: The MOST Workbench.* In Pan, J. Z., Staab, S., Aßmann, U., Ebert, J., and Zhao, Y., editors, Ontology-Driven Software Development, pages 275-292. Springer Berlin Heidelberg.

- Aßmann, U., Zivkovic, S., Miksa, K., Siegemund, K., Bartho, A., Rahmani, T., Thomas, E., and Pan, J. (2013). *Ontology- guided Software Engineering in the MOST Workbench.* In Pan, J. Z., Staab, S., Aßmann, U., Ebert, J., and Zhao, Y., editors, Ontology-Driven Software Development, pages 293-318. Springer Berlin Heidelberg.

- Ren, Y., Grüner, G., Rahmani, T., Lemcke, J., Friesen, A., Zivkovic, S., Zhao, Y., and Pan, J. (2013). *Ontology Reasoning for Process Models.* In Pan, J. Z., Staab, S., Aßmann, U., Ebert, J., and Zhao, Y., Editors, Ontology-Driven Software Development, pages 219-252. Springer Berlin Heidelberg.

- Bartho, A., Grüner, G., Rahmani, T., Zhao, Y., and Zivkovic, S. (2011). *Guidance in Business Process Modelling.* In Service Engineering: European Research Results, pages 201-231. Springer Vienna.

- Schwarz, H., Ebert, J., Lemcke, J., Rahmani, T., and Zivkovic, S. (2010). *Using Expressive Traceability Relationships for Ensuring Consistent Process Model Refinement.* In Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on, pages 183-192. IEEE.

- Zivkovic, S., Kühn, H., and Murzek, M. (2009). *An Architecture of Ontology-aware Metamodelling Platforms for Advanced Enterprise Repositories.* In Proceedings of the 1st International Workshop on Advanced Enterprise Repositories (AER 2009), Colocated with 11th International Conference on Enterprise Information Systems (ICEIS 2009) Milano, Italy, May 6th, pages 95-104.

- Zivkovic, S., Murzek, M., and Kühn, H. (2008). *Bringing Ontology Awareness into Model-driven Engineering Platforms.* In Parreiras, F. S., Pan, J., Aßman, U., and Henriksson, J., editors, Proceedings of the 1st International Workshop on Transforming and Weaving On- tologies in Model Driven Engineering (TWOMDE 2008), Co-located with MODELS, Tolouse, France, September 28, 2008, CEUR Workshop Proceedings, pages 47-54. CEUR-WS.org.

Part II

Existing Work

Chapter 2

Concepts for Modelling Method Engineering

"Essentially, all models are wrong, but some are useful."

George E. P. Box

Modelling method engineering represents the conceptual umbrella field of the underlying work. Modelling method engineering deals with design, construction and adaptation of modelling methods and appropriate tools for model-based system analysis and development. Especially the engineering of situational, hybrid and evolving modelling methods is of interest, in order to support the more general idea of Agile Modelling Method Engineering (AMME) [Karagiannis, 2015]. On the one side only such methods nowadays can flexibly satisfy the increasing pace of business requirements for appropriate project-, situation-, domain-specific modelling methods and tools. On the other side, engineering of such methods requires rethinking in engineering concepts, techniques and tools in terms of modular and compositional engineering, which is the main problem area to be advanced. Although, in this work, the special focus is on metamodels, which represent the central, pivotal part in the definition of modelling languages and, thereby, of modelling methods, in this chapter we provide a broader overview of the existing work on modelling method engineering.

The chapter is organised as follows. In Section 2.1, we introduce the basic concepts related to the existing work on engineering of modelling methods in general. After that, in Section 2.2, the focus is set on the method modularisation and method integration approaches for hybrid modelling methods, a modelling approach referring to multi-perspective, coordinated use of multiple methods. Finally, in Section 2.3 three case studies related to engineering of hybrid modelling methods based on existing method integration approaches are provided. Section 2.4 summarises the chapter.

2.1 Modelling Methods

A method is the way of accomplishing a goal by turning some input into an output/result. In the field of information systems engineering, method engineering focuses on procedures, principles, languages and tools to support the development of methods. As a broader term within the computer science, method engineering is an engineering discipline that deals with design, construction and adaptation of methods and tools for system development [Brinkkemper, 1996]. Situational method engineering (SME) [Harmsen, 1997], a special domain of method engineering, concentrates on the engineering of situation-specific, project-specific methods. The situational aspect advocates considering organisational, project requirements and characteristics when constructing a method, instead of using a general one-size-fits-all solution. Based on various studies in organisations on method acceptance and use, the phenomenon of favorising custom solutions over standards has been described by Tolvanen as a paradox in ISD methods [Tolvanen, 1998]. In situational method engineering, the method construction process starts by defining method requirements for a current situation, selecting method components satisfying this situation and assembling the selected method components. A special aspect of SME focuses on an assembly of existing method fragments (method chunks) to create domain specific solutions [Ralyté et al., 2006]. Specific to modelling method engineering, Karagiannis introduces an approach to agile modelling method engineering (AMME) [Karagiannis, 2015]. Inspired by agile principles in software engineering, AMME tackles the challenge of continuously changing modelling requirements, and the necessity for more flexibility in engineering of modelling methods.

2.1.1 Elements of a Method

Common Method Elements

Typically, a method consists of a *product part* and a *process part* [Rolland et al., 1999]. According to [Mirbel and Ralyté, 2005], the product model of a method defines a set of concepts, relationships between these concepts and constraints for a corresponding schema construction, whereas the process model describes how to construct the corresponding product model. The process part prescribes the way how artefacts are used to produce specific method outputs. A method may consists of one or more product models and one or more process models. Figure 2.1 summarises the method definition. The process and the product part of a method can be undoubtedly recognised in the method definition of Booch [Booch, 1991]. According to Booch, a (modelling) method is a rigorous process allowing to generate a set of models describing different perspectives of the system under construction by using some well-defined notations. That said, the product part describes

artefacts used in the method. The set of models described using a dedicated notation may be regarded as a product part, whereas the rigorous process refers to the process part. Henderson-Sellers criticises the common method definition of having process and product parts by pointing an often-neglected third *people* aspect [Henderson-Sellers and Ralyté, 2010]. While a development method consists of some development activities (process) that are used to create some work products (product), he argues that producers (humans or machines) are the third component that executes the process to produce products. However, producers are often considered within the process part as roles responsible for executing activities.

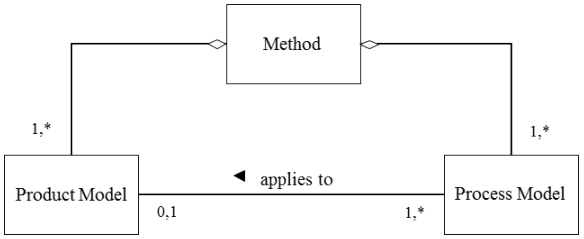


Figure 2.1: Elements of a method

Elements of Modelling Methods

Another viewpoint on method definition, which concentrates in particular on the methods used for modelling, i.e. modelling methods, is introduced by [Karagiannis and Kühn, 2002]. In their seminal work, Karagiannis and Kühn argue that a modelling method consists of two main elements 1) modelling technique and 2) mechanisms and algorithms. Analog to the basic program definition which consists of data structure and algorithms by Wirth [Wirth, 1978], the modelling technique defines the structure upon which computation may operate in terms of mechanisms and algorithms. Modelling technique itself is further divided into a modelling language and modelling procedure. The product part corresponds to the modelling language, and the process part to the modelling procedure. In a nutshell, within a method, modelling languages are used to create models according to a defined modelling procedure, upon which algorithms and mechanisms operate. By emphasising the importance of mechanisms and algorithms, the approach by [Karagiannis and Kühn, 2002] focuses on modelling methods and its tool-supported operationalisation. While some organisational methods might not have any specific mechanisms and algorithms, language-centred modelling methods, in turn, usually do come with a set of various tool-supported mechanisms and formal algorithms that enrich and facilitate the method use in reaching project-specific method goals. Mechanisms and algorithms operate on models created using the modelling language.

Mechanisms and algorithms contribute to modelling methods by adding the processing aspect of a method. This third aspect of modelling methods is often partly or completely neglected in other works. However, it is regarded as crucial, since it generates an added value beyond model-based system capturing, especially for tool-supported modelling methods. It needs to be mentioned that modelling methods found in literature usually do not contain all elements of a full modelling method. Usually, modelling method term is used to describe a modelling language (notation) such as UML, BPMN. However, if we take the software development domain/methodology, UML may be regarded as a modelling language, whereas parts of UP [Jacobson et al., 1999] that refer to modelling may be considered as a matching modelling procedure. As for mechanisms and algorithms, an example of a mechanism may be a code generator that transforms a UML class diagram to a program code. An example for an algorithm may be an Hopcroft's optimisation algorithm [Hopcroft, 1971] for computing the minimal finite state machine that may be applied for UML state diagram validation and optimisation.

Figure 2.2 illustrates the framework for modelling methods. A detailed description of this framework in its extended form is provided in the following section.

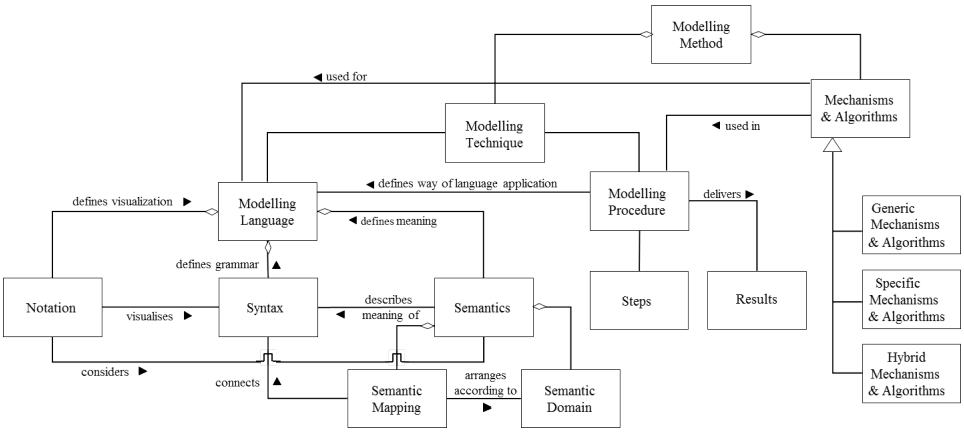


Figure 2.2: Modelling method framework according to [Karagiannis and Kühn, 2002]

2.1.2 Extended Method Framework

The framework by Karagiannis and Kühn focuses on modelling methods for systematic tool-supported model-based system development (analysis, design and implementation). Its fundamental part, the modelling technique, also viewed as a static part of the method, is enriched by mechanisms and algorithms, referred to as a dynamic part of the method. When compared to

the broader definition of methods for situational method engineering [Rolland et al., 1999, Mirbel and Ralyté, 2005, Henderson-Sellers and Ralyté, 2010] introduced earlier, modelling technique, in particular, modelling language corresponds to the method product part, whereas the modelling procedure may be considered as a process part. However, while modelling procedure provides guidelines how to create complete and correct models using the language, it doesn't specify the overall temporal view on the method. By that, a kind of development methodology is meant, which specify when, how and by whom the models i.e. method products are created. If we again consider the UML as modelling technique, the UP may be considered as process, that specifies the whole software development methodology. [Kühn, 2004] extends the modelling method framework introduced by [Karagiannis and Kühn, 2002] by adding the third missing process aspect of a method.

According to it, a method is a triple of:

- a modelling technique (language),
- mechanisms and algorithms, and
- a process model.

Figure 2.3 illustrates the extended view on the modelling method framework. Each of those elements are explained in detail in the following sections.

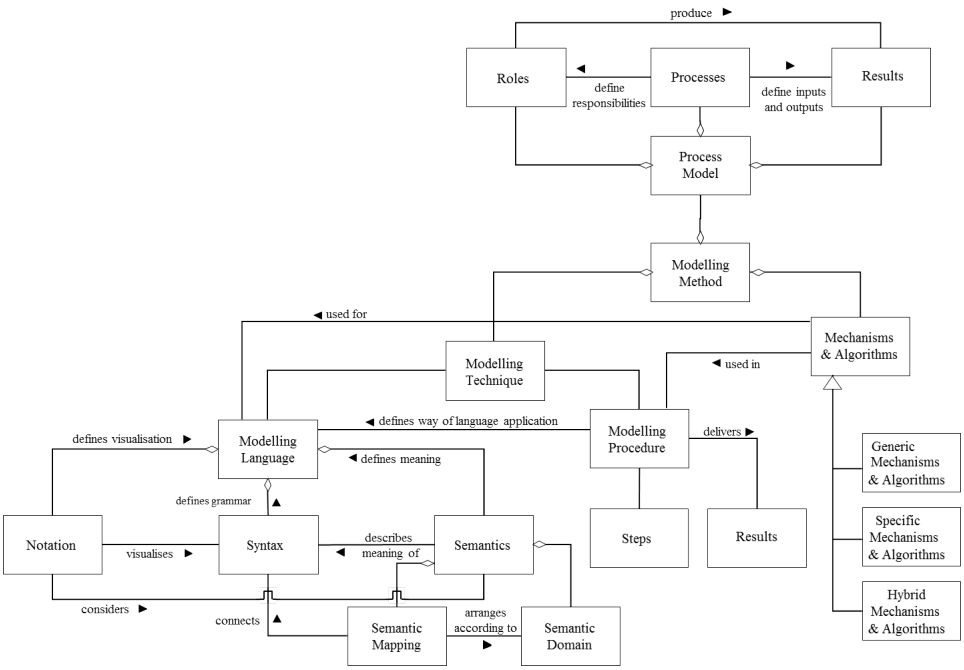


Figure 2.3: Extended modelling method framework based on [Kühn, 2004]

2.1.3 Modelling Technique

Modelling technique defines the modelling language of a method and provides guidelines and best practices for its use.

Modelling Language

Modelling language represents the structural part of a modelling method. It defines the formalism to capture the data about the system under consideration in terms of models. Modelling language is a pivotal element of a method around which, mechanisms and algorithms are built and for which modelling procedures are advised. According to the modelling method framework, modelling language itself is described by syntax, semantics, and notation. The syntax defines the grammar of the language. Semantics defines the meaning of the language, by mapping the syntax elements to their semantic counterparts in the semantic domain/schema. Semantic schema may be formal or defined using informal language such as textual descriptions of syntactic concepts. The notation is about the language visualisation. Each of the syntax elements receives a visual representation symbol. In textual modelling languages these may be single language keywords, in graphical modelling languages these may be graphical symbols. Note that syntax may have more than one notation. For example, a business process model may be visualised as a graphical model but also as a textual description. The language part of a method is of special interest of the underlying work. Therefore, a deeper discussion about modelling languages is provided in the Chapter 3.

Modelling Procedure

Modelling procedure advises the steps on how to correctly use the modelling language as well as the expected results of the language use, in terms of models. Modelling procedure provides a kind of direct guidance for modelling. According to [Kühn, 2004] a modelling technique should provide support for the following aspects of modelling: 1) completeness, 2) correctness, 3) adequacy, 4) complexity (reduction). While the underlying modelling language fundamentally influences the adequacy aspect, modelling procedure in terms of guidance can influence the completeness, correctness and complexity reduction of models to a great extent. Guidelines and best practices on how to model with decomposition in mind to reduce the complexity or rules combined with syntax and static semantic constraints towards completeness and correctness of models are examples of modelling procedure applications. Also, the usage of adequate modelling constructs for a specific problem under consideration can be captured in terms of best practices and patterns within the modelling procedure. An attempt to automate modelling guidance in

combination with model validation in the field of model-driven software development and process modelling has been extensively elaborated in the work of [Aßmann et al., 2013] and [Bartho et al., 2011]. There, a semantic technology-based process guidance engine has been developed that guides users through the modelling tasks by pointing out errors and providing hints on how to correct them.

2.1.4 Mechanisms and Algorithms

As an extension to the common agreement on method structure, Karagiannis and Kühn [Karagiannis and Kühn, 2002] propose mechanisms and algorithms as a third basic element of methods. Although mechanisms and algorithms as terms are often used together i.e. interchangeably, there is a difference in meaning between mechanisms and algorithms worth mentioning. According to Wiki the term algorithm in mathematics and computer science refers to a step-by-step procedure for calculations. Formal or informal an algorithm is a specification for some kind of calculation on data. For example, the famous Dijkstra's algorithm ([Dijkstra, 1959]) is a graph search algorithm that solves the single-source shortest path problem for graph-based data structures. Mechanisms (in Greek "machine") is any kind of system or machinery that executes some instructions (computer, human). In relation to algorithm, mechanism is a system that is able to execute an algorithm (e.g. a Turing machine). Mechanism is a kind of algorithm implementation/operationalisation. That said, algorithms in modelling methods are used for calculation, processing and reasoning on model data that are run by various mechanisms. For example business process management method BPMS provides a simulation algorithm which is realised as a process evaluation mechanism within the business process modelling tool ADONIS.

Mechanisms and algorithms may be defined using different formalisms depending on the target purpose. These may range from informal natural languages, pseudocode, to more formal notations such as flow charts, state charts. When it comes to their implementation for modelling tools, machine-interpretable imperative and declarative programming and scripting languages are used.

Types of Mechanisms and Algorithms

There are various kinds of mechanisms and algorithms as well as different ways how to classify them. According to [Kühn, 2004] mechanisms and algorithms for modelling methods can be categorised according to the *abstraction level* on which they are defined. In addition, we introduce a new category for classifying mechanisms and algorithms based on *application purpose*. In the following we discuss these two categories.

Categorisation Based on Abstraction Level

- *Generic mechanisms and algorithms.* Generic mechanisms and algorithms are independent of the underlying modelling technique in use, thus making them applicable for any kind of modelling technique. This is achieved by specifying an algorithm on the level of meta-metamodels instead of metamodels. An example of a generic mechanism is usually an import/export mechanism for model data exchange, that doesn't require any modelling language specific semantics.

- *Specific mechanisms and algorithms.* Specific mechanisms and algorithms are bound to the semantics of one modelling language. They are thus defined on the level of metamodels for one specific modelling language. An example of a metamodel-specific mechanism is process simulation, which operates on models of one concrete process modelling language (e.g. BPMN).

- *Hybrid mechanisms and algorithms.* Hybrid mechanisms and algorithms are defined in a generic manner, with an option to either configure them or adapt them to one or more specific modelling languages. While Kühn [Kühn, 2004] doesn't differentiate the hybrid category any further, it may be noticed, that hybrid mechanisms may further be divided into *configurable* and *adaptable* mechanisms.

  - *Configurable mechanisms.* Configurable mechanisms are basically like generic ones, but with specific configuration that is applicable for any kind of modelling languages. An example of a configurable mechanism may be the model transformation. While model transformation is realised on the meta-metamodel level (e.g. a transformation rule requires a source and a target class) it can be configured to a specific modelling language (e.g. a class A transforms to class B) and applied on models of that particular language.
  - *Adaptable mechanisms.* On the other side, adaptable mechanisms are bound to specific semantics of a family of modelling languages (e.g. process modelling languages). Thus, they are more restrictive, but still more generic than specific mechanisms, since they may be adapted to a concrete modelling language. An example of such mechanism may be simulation that is bounded to an abstract process metamodel, that by derivation may be adapted to a particular one. In [Prackwieser et al., 2013] a hybrid simulation algorithm is proposed that, while generic, may be applied to a chain of business process models based on multiple notations.

Categorisation Based on Application Purpose

- *Analytical mechanisms and algorithms.* Analytical mechanisms and algorithms are used for model data analysis. They do not change any model data, but just read and help to better understand complex model structures. Typical examples are model querying, model search, model comparison and model validation mechanisms and algorithms. An example of a language specific validation is a process refinement consistency checking mechanism for BPMN models ([Ren et al., 2009]).

- *Computational mechanisms and algorithms.* Computational mechanisms and algorithms are used for any kind of computations on model data. The calculation is based on existing data, based on which new aggregated model data is generated. Process model simulation is an example of a complex computational mechanism. Another example is the risk assessment, used in the area of enterprise risk management, where risks are evaluated based on characteristics such as likelihood of occurrence and magnitude of risk impact [COSO, 2013]. Such mechanisms are part of model-based enterprise risk management methods (BPMS) and tools such as ADONIS.

- *Generative mechanisms and algorithms.* Generative mechanisms and algorithms generate new (model)data out of models. Model transformation, a key mechanism for MDE ([Kent, 2002], [Kleppe et al., 2003]) is a generative mechanism, which transforms source models to target models or to text based on well-defined mappings. There is a major distinction between model-to-model and model-to-text approaches. Model-to-model transformation is a function that receives a source model, a source metamodel, a target metamodel and a set of transformation rules as input and produces a target model which conforms to a target metamodel. A thorough analysis of different approaches for model transformation is provided in [Czarnecki and Helsen, 2006].

- *Manipulative mechanisms and algorithms.* Manipulative mechanisms and algorithms manipulate i.e. change the existing model data. Various types of model editors belong to this category. Furthermore, model merge is a manipulative model mechanism which integrates two models into one based on merging operators [Pottinger and Bernstein, 2003], [Brunet et al., 2006].

Table 2.1 collects typical mechanisms and algorithms of modelling methods and classifies them according to the above mentioned categories.

Table 2.1: Classification of typical mechanisms and algorithms of modelling methods

	Generic	**Hybrid**		**Specific**
		Configurable	**Adaptable**	
Analytical	model comparison	model checking (OCL), model traceability, business impact analysis	consistency checking for structural models, consistency checking for process models	consistency checking of BPMN process refinements, verification of UML state charts [David et al., 2002]
Computational	model metrics		simulation of process models	simulation of petri nets, risk management assessment(BPMS)
Generative	model exchange (import, export)	model transformation	code generation for lang. families	BPMN2BPEL transformation, lang. specific code generation (UML2Java, BPMN2BPEL)
Manipulating	model editors (textual, graphical, tabular), object replacement	model merge, object ordering, model alignment, class conversion, model versioning		optimisation of state charts

2.1.5 Process Model

Process model prescribes steps, results and roles necessary to achieve the goals of a modelling method by using the modelling language. Process model represents the process part of a modelling method. For example, as part of a holistic methodology for software development, unified process (UP) [Jacobson et al., 1999] prescribes how software may be modelled using the software modelling language UML [OMG, 2011b]. In particular, how different diagram types may be employed to model the software system from different perspectives. While the necessity of having a process model for small modelling languages (one diagram type) is arguable, complex modelling frameworks such as TOGAF [The Open Group, 2012], BPMS [BOC, 2015], can be hardly efficiently used without a proper process model (also

known as framework or methodology).

Process Views

According to [Kühn, 2004], there are three basic views of the process model: 1) process view, 2) actor view 3) result view. The process view is concerned with milestones, phases, tasks/activities and steps and their semi-ordered execution. The actor view is about roles that execute processes to produce results. Finally, the result view defines the result types, typically models, and represents the connection to the modelling language part of the method. Similarly, [Henderson-Sellers and Gonzalez-Perez, 2005] compare various process metamodels and identify three main aspects of software development processes: work units, work products and producers. Essentially, work units are performed by producers that consume (input) and produce (output) work products. In the case of software development processes, work units are abstractions of tasks, activities, steps but also builds, phases, milestones, cycles, etc. Producers are often represented by the users which may have different roles. Producers can also be other tools, that are enacted to perform certain work units. The work products are development artefacts. These can be source code files, requirement documents or models. While work units correspond to the process view, producers are described in the actor view, and work products in the result view. Figure 2.4 illustrates different process views by emphasising their intersections.

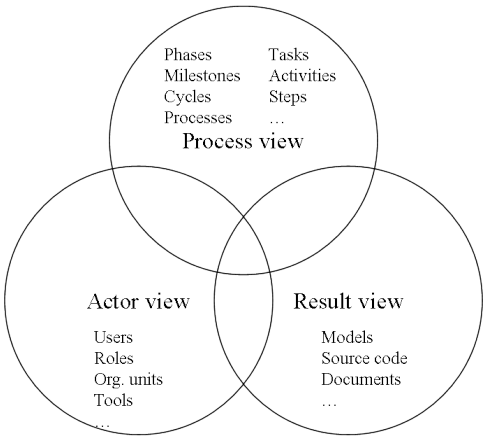


Figure 2.4: Process views

There is a multitude of standardised process models in different engineering areas. Usually, such process frameworks are customised for specific purposes. For example, In [Kühn et al., 2011] a development process for a method for model-driven development of interoperable business processes is suggested. Figure 2.5 illustrates the specification of the method process

part. Note how different views of the process model are considered and how they correlate.

Phase	Step/Objective	Actor	Language	Result
Strategic	• Define main objective • Provide elements of evidence • Design business model • Common understanding of inter-organisational business processes	• Top-level management • Business experts	• Business Model (BPMS), • Cooperation pictures (BPMS)	• Business model • Feasibility study
Operational	• Analyse / create business process models • Identify business interfaces and documents for the data exchange • Align models between business partners • Achieve detailed understanding of processes and documents	• Operational management • Business analysts	• Business Process • Model (BPMS) • Event-driven process chains (EPC) • UML activity diagram	• (Aligned) business process models • Business interfaces
Development	• Transform business process models to BPMN models • Refine BPMN models • Workflow and document description	• IT architects • Application developers	• BPMN • Business information model • CCTS • ER • XPDL	• BPMN models • Document description • Integrated process and data models
Execution	• Transform BPMN models to BPEL / WSDL • Enrich, test, deploy, and execute the business process	• Application developers • IT operators	• BPEL • WSDL • XPDL • XML • XSD	• Executable business process definitions (BPEL) • Executable business documents

Figure 2.5: Example of a process model. Development process specification for model-driven development of interoperable, inter-organisational business processes [Kühn et al., 2011]

Specification of Process Models

Process models may either be specified informally using textual descriptions (e.g. manuals) or formally using some kind of a process modelling language. For example, in the area of software and systems process methodologies, the standardized software and systems process engineering metamodel (SPEM 2.0, [OMG, 2008]) may be used as a formalism to specify arbitrary software processes and best practices. SPEM is not only a formal metamodel but also a conceptual framework which provides necessary concepts for modelling, documenting, presenting, managing, interchanging, and enacting development methods and processes. Due to its generic nature, SPEM may be applicable as a formalism for other system development methods, too.

Figure 2.6 illustrates an expert of the the SPEM 2.0 metamodel focusing on the process part of the process model. A central concept for defining a process or a work breakdown structure and sequence is an *activity*. Activities may be comprised out of other activities and may have connections to other process model aspects such as the actor view and the result view. For example, a *milestone* defines required results in form of *work products*. On the other side, work definitions such as Activities are executed by *process performers* that may have different *roles* and *responsibilities* assigned. With SPEM 2.0 and its work breakdown-like process structure definition, it is possible to define different types of processes such as waterfall, iterative, incremental, etc.

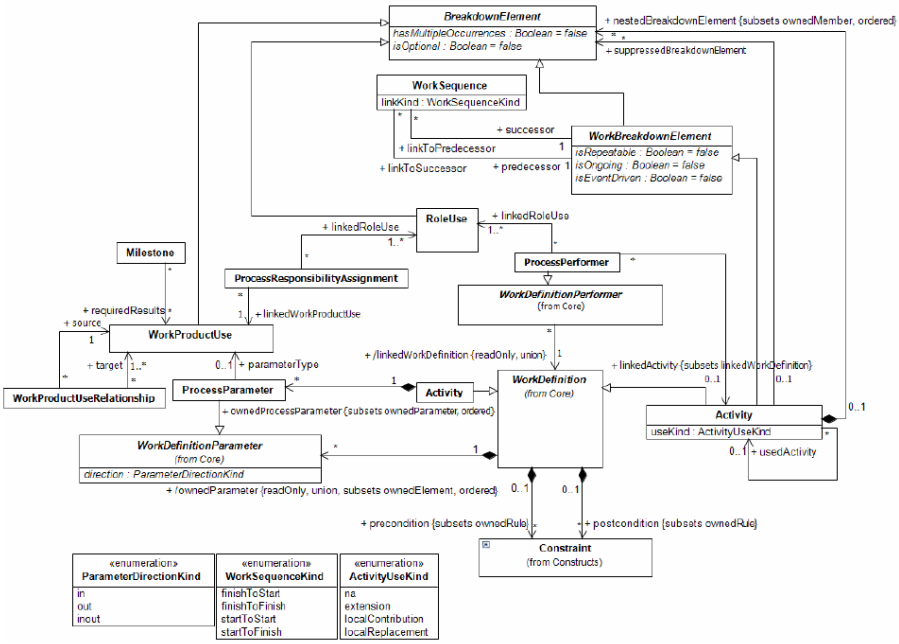


Figure 2.6: Process structure part of the SPEM 2.0 Metamodel (SPEM 2.0, [OMG, 2008], p.44)

2.1.6 Overview of Method Engineering Approaches

Method engineering is about choosing an appropriate method for the problem at hand. Basically, method engineers may need to create a new method from scratch, modify (incrementally improve, tailor or constrain) an existing method, or reuse parts of various methods and compose them into a new method, or any combination of the above. That said, two major approach categories may be differentiated: 1) from scratch approaches, 2) reuse-based approaches. In [Ralyté et al., 2003] four different method engineering approach categories have been identified:

- *Ad hoc* (from scratch) approaches,
- Paradigm based approaches,
- Extension-based approaches,
- Assembly-based approaches.

The *ad hoc* approaches start developing methods from scratch. Several techniques exist that improve the *ad hoc* method construction process towards higher reuse. The other three approaches rely on a certain degree of reuse, out of which assembly-based approaches exhibit the highest level of reuse. In the following, these four approaches are further discussed. Figure 2.7 illustrates the classification of method engineering approaches.

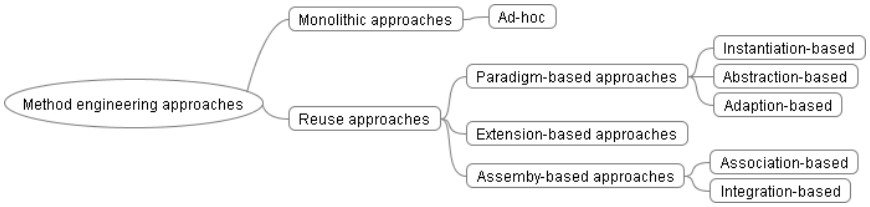


Figure 2.7: Topology of method engineering approaches (adapted from [Ralyté et al., 2004]

Ad hoc Approaches (From Scratch)

Ad hoc approaches start building methods from scratch. While this kind of method construction has the biggest adaptation freedom (100%), it also has, due to zero reuse potential, the highest development efforts (while building from scratch). Nevertheless, developing methods from scratch is a valid approach when creating methods for new domains, for which no existing methods or reuse elements exist yet.

Paradigm-based Approaches

Paradigm-based approaches rely on explicated knowledge reuse. According to [Ralyté et al., 2004], a paradigm may be a model, a meta-model as baseline As-Is model which is then instantiated, abstracted, or adapted to create a To-be model, the basis of the method. A paradigm may be referred to as a kind of pattern or a reference model for a certain domain. The authors also point out the difference in abstraction level of As-Is and To-Be models. In the case of adaptation, the As-Is and To-Be models are on the same level of abstraction, whereas in the cases of instantiation and abstraction, the abstraction level of As-Is and To-Be models is different. In [Becker et al., 2009], an example of a method construction by adaptation is given, in which

a business process modelling language is adapted specifically for the needs of banks based on the process patterns for the analysis of processes in the banking sector. An example of an As-Is paradigm may be Petri nets, based on which, various modelling methods and notations have been constructed such as UML activity diagrams [OMG, 2011b], BPMN [OMG, 2013] and EPCs [Keller et al., 1992].

Extension-based Approaches

Extension-based approaches adapt methods based on extensions. Extension mechanism for tailoring is recognised as a viable approach to method engineering. There is a number of modelling and software development methods that favour extension-based tailoring such as RUP [Kruchten, 2004]. The prerequisite for tailoring/extension-based approach is an already existing method that is then adapted to the problem/situation at hand. Such base method should be generic enough to be applicable for number of problems through customisation. Usually, such methods are kind of generic frameworks such as UP [Jacobson et al., 1999], or TOGAF ADM [The Open Group, 2012]. The more abstract the method is, the more customisation freedom is given with the price of more customisation effort. On the other side, the more specific the base method is, the less freedom for customising exists, and the less customising effort is required. Another example for an extension-based method construction may be found in the domain of formal process modelling. Interestingly, besides paradigm-based method construction, petri nets have been used also for extension-based method construction. While petri nets represent a highly generic formalism with very broad application fields, several extensions have been proposed, such as coloured, hierarchical and timed petri nets. Furthermore, UML ([OMG, 2011a], [OMG, 2011b] which may be extended via the profile mechanism featuring stereotypes [Fuentes-Fernández and Vallecillo-Moreno, 2004], belongs to the family of extension-based approaches.

Assembly-based Approaches

In assembly-based approaches, methods are constructed on-the-fly, in order to match as much as possible to the situation of the project at hand, based on existing method components. SME introduces the concept of method fragments/chunks. There are three important aspects of SME with MEs. 1) method fragment definition 2) storage, retrieval, repository of method fragments 3) assembly techniques. Assembly-based approaches foster reuse and thus reduce the effort in designing and maintaining methods, while keeping the adaptation freedom at the higher level. The idea of developing (modelling) methods, as broader concept, by combining the existing method parts/fragments has been proposed by Brinkkemper [Brinkkemper

et al., 1999] to build tailored situational methods. Here, not only metamodels are combined (method product part), but also process models as method process parts. Following this idea, Ralyte [Ralyté, 1999, Mirbel and Ralyté, 2005] introduces method chunks, reusable method building blocks. Having fragments and chunks for method modularisation, techniques for their combination are equally important. Two basic assembly strategies have been identified, association-based and integration-based [Ralyté and Rolland, 2001, Mirbel and Ralyté, 2005]. Assembly-based association techniques as defined in [Brinkkemper et al., 1999] are applied in cases when method fragments are complementary. Method fragments are associated by defining links between their concepts. On the other side, assembly-based integration techniques [Ralyté et al., 2004] are applicable in cases where method fragments are overlapping. In such cases, chunks are merged into a new richer chunk. Abstracting from a specific approach, Kühn [Kühn et al., 2003,Kühn, 2004] identifies the most common method integration techniques as conceptual patterns and proposes a pattern system for method integration. In [Zivkovic et al., 2007] another assembly-based approach is proposed that offers an integration algebra in terms of mappings and integration rules for integrating method product parts (metamodels).

The Method Engineering Continuum

When observing the evolution of different method engineering approaches, it becomes clear that two main principles drive the advancement in the field of method engineering, the principle of reuse and the principle of adaptation (to satisfy user requirements). The principle of reuse leverages increased quality and productivity while reducing engineering efforts. The principle of adaptation allows for problem-specific method tailoring.

Figure 2.8 illustrates using a portfolio chart the comparison of the previously introduced approach categories with regard to reuse, adaptation freedom and engineering efforts. While ad-hoc approaches exhibit the highest level of adaptation freedom, they do not support reuse. Consequently engineering effort is very high. Paradigm-based approaches reuse a certain paradigm to start the construction (e.g. via metamodel or pattern instantiation, model abstraction or reference models), which leads to lower engineering effort. By doing this, the adaptation freedom decreases slightly, but remains on the high level, since reused concepts may be adapted. Extension-based approaches rely on the reuse of complete base methods or method frameworks, thus significantly reducing the engineering effort. On the other side, the adaptation freedom is considerably constrained by the given framework. Finally, the assembly-based approaches combine reusable components for method construction, showing the highest level of reuse with the lowest engineering effort in comparison to the previous approaches. It is worth noticing that for assembly-based approaches the adaptation freedom curve

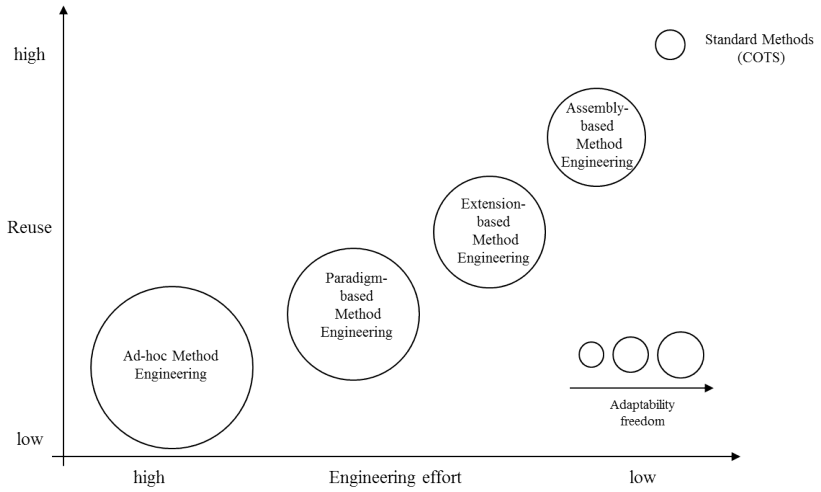


Figure 2.8: The method engineering continuum

again raises. This phenomenon may be attributed to the flexible selection of existing reuse components. Although standard methods are not the focus of situational method engineering due to the one-size-does-not-fit-all syndrome, it is worth including this approach as well for the comparison. This reveals, under the assumption that a standard method does perfectly match requirements of the problem at hand (a perfect method), that standard methods have the absolute reuse level (reused as whole) with no (additional) engineering efforts, however offering no space for adaptation.

2.2 Method Integration for Hybrid Modelling

In this section, we introduce the notion of hybrid, integrated modelling methods. By doing so, we introduce different categories of hybrid modelling methods, provide an overview of some existing methods and discuss the role of modelling frameworks for hybrid methods. As mentioned before, assembly-based techniques focus on the integration of existing method fragments to build new hybrid methods. Therefore, we discuss method (fragment) modularisation as an important aspect in developing hybrid methods. Having modularised method fragments, new hybrid methods may be created by fragment integration. Hence, we provide an overview of method (fragment) integration i.e. assembly approaches such as pattern-based, mapping-based and change-centric based integration. We finalise the section by discussing the overall life cycle of hybrid modelling methods.

2.2.1 Hybrid Modelling Methods

Similar to the various architectural views of the physical building, a complex business or software system is viewed and analysed from different viewpoints/perspectives to better understand and specify its structure and behaviour. Multi-perspective coordinated use of multiple modelling methods is regarded as a holistic, systematic approach to model-based system specification, known as hybrid modelling.

Classification of Hybrid Modelling Methods

Modelling methods may vary based on the system aspect they are describing or based on the granularity/abstraction level at which they describe certain aspect. Method or method components describing different system aspects are orthogonal, complementary to each other, also horizontal. On the other side, method or method components describing the same system aspect but at different levels of abstraction are said to be vertical, refinement methods.

The integration of modelling methods into hybrid approaches may consider composition of method components from different aspects and/or abstraction levels or any kind of combination of those dimensions. Based on the specific combination of method components, three basic integration approaches may be distinguished (see also [Kühn et al., 2003]). We add an additional composite category which consists of combinations of basic integration approaches.

- *Horizontal integration.* The horizontal integration integrates orthogonal languages that reside on the same abstraction level. This is usually the case for hybrid languages that need to support more than one system aspect. This approach may be called method broadening, since the method is extended. An example of a horizontal integration is the extension of the BPMN with organisational data modelling. According to the literature research (see subsequent section), this kind of hybrid languages is the most common one. There is one additional scenario where two languages are not complementary but overlapping. However, this kind of horizontal integration is not further discussed due to its questionable usefulness.

- *Vertical integration.* The vertical integration combines languages that describe the same system aspect/domain but at different levels of abstraction. This kind of hybrid languages is typical for methods that follow step-wise refinement modelling procedures. For example, BPMN modelling language is used to model detailed business processes on the technical level. These process descriptions may be rewritten and refined on the execution level using workflow languages such as BPEL. The vertical integration is known as refinement if done in top-down

manner, or generalisation in case of bottom-up, reverse engineering approaches.

- *Cross-wise integration*[1]. The cross-wise integration combines the horizontal and vertical integration approaches. Languages describe different system aspects but at different levels of abstraction. This kind of hybrid languages may occur when for example only one kind of aggregated language is used on the higher abstraction level, which is refined to various system aspect languages on the lower abstraction level. For example, a business process modelling language may connect to lower level languages to acquire run-time data for process simulation.

- *Multi-dimensional integration.* Multi-dimensional integration is a composite integration approach in which several methods are integrated combining vertical, horizontal and/or cross-wise integration dimensions. This is a typical case for modelling frameworks.

Figure 2.9 illustrates the framework for hybrid languages according to integration dimensions.

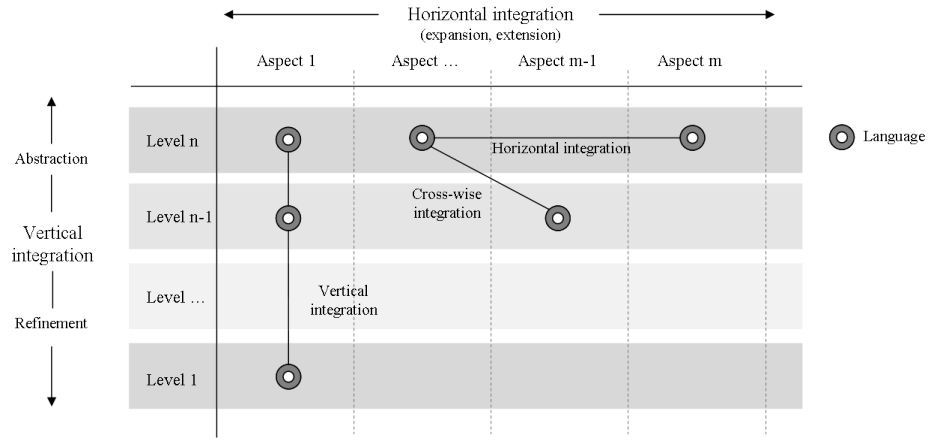


Figure 2.9: Framework for hybrid languages and integration dimensions

Modelling Frameworks

Modelling frameworks provide an architectural basis, a frame to develop systematic, holistic, hybrid languages. Usually, a framework suggests a concrete development process and a language framework with or without a specific set

[1]In [Kühn et al., 2003], Kühn et al. use the term "hybrid" to depict this kind of integration approaches. However, since this term has been coined to describe any kind of combined, integrated approaches, an alternative term has been introduced.

of modelling languages. Ideally, the language framework is "filled" by concrete languages according to the organisational, problem and project specific needs. Two areas of modelling frameworks emerged over time in the field of information system development 1) frameworks for modelling enterprise architectures (Zachman [Zachman, 1987], ARIS [Scheer, 1992], GERAM [IFIP-IPAC Task Force, 1999], E-BPMS [Kühn et al., 2001], MEMO [Frank, 2002], CIMOSA [Vernadat, 2006], TOGAF [The Open Group, 2012] with ArchiMate [The Open Group, 2013]), BPMS [BOC, 2015] and 2) frameworks for software modelling (UP [Jacobson et al., 1999] with UML [OMG, 2011b]).

In the context of hybrid modelling methods, modelling frameworks are useful as they may provide a reference or a starting point to decide which aspects, concerns and levels of abstraction should be covered by a modelling method. For the realisation of a hybrid method, usually all kinds of integration approaches are pursued: horizontal integration, vertical integration and cross-wise integration.

Overview of Hybrid Modelling Methods

Nowdays, a multitude of hybrid modelling methods in research and practice exist. The approaches range from integrating "only" the product part of a method, i.e. the modelling technique, to those that integrate process part as well. In the following, some examples of hybrid modelling methods found in literature are briefly introduced.

- In [Shen et al., 2004] a hybrid modelling method for enterprise information system analysis and user requirements gathering is proposed. The proposed hybrid method integrates three modelling methods. Whereas IDEF0 is used to create functional models, IDEF3 captures process descriptions. Finally, DFD focuses on information/data flow between process activities. The authors also propose an integrated/hybrid modelling procedure on how to use the composite method.

- In [de Kinderen et al., 2012], the ArchiMate modelling language for enterprise architectures is integrated with the value modelling language e^3value. For integrating the metamodels the authors used the metamodel integration approach proposed by [Zivkovic et al., 2007].

- In [Morin et al., 2009] a hybrid modelling method approach is suggested, that integrates variability aspect into arbitrary domain-specific modelling languages (DSML) that are based on Eclipse based metametamodel Ecore/EMOF. The application domain is a hybrid modelling of software using a domain-specific language and feature diagrams to facilitate product-line based software development.

- In [Xu et al., 2010], [Tan and Liu, 2012] a hybrid modelling method Active i* is proposed that combines the i* method with UML activ-

ity diagrams. An agent-oriented requirement modelling language, i*, is adopted to illustrate the high-level business objectives of organisational units, while UML activity diagrams are used to represent the business activities in the production process.

- In [Rausch et al., 2011] the standard business process modelling notation BPMN has been extended with business relevant concepts to support systematic business analysis. By doing this, the BPMN notation is combined with the ADONIS Business Process Management Systems Method (BPMS, [BOC, 2015]) to support modelling of multiple business aspects such as processes, organisation, products and services and IT. For metamodel integration, the authors follow the metamodel integration approach proposed by [Zivkovic et al., 2007].

- In [Schroth et al., 2007], a hybrid modelling method for business information modelling is proposed that enhance business process modelling with data modelling based on CCTS standard. For business processes on business level ADONIS BPMS is used, whereas BPMN is used to describe the technical process workflows. For the data model description the CCTS standard (developed by UN/CEFACT) has been integrated, which provides enhanced data modelling capabilities such as contexts for electronic business documents.

- In [Zivkovic et al., 2007], a case study on a hybrid modelling method for business process modelling based on extended BPMN is provided. The BPMN is horizontally extended by organisational modelling concepts from the ADONIS BPMS method. The integration is based on the metamodel-based integration approach.

- In [Gérard et al., 2011], an integrated language for real-time and embedded systems, MARTE, is introduced. The MARTE language is implemented as a UML profile, enables precise modelling of phenomena such as time, concurrency, software and hardware platforms, as well as their quantitative characteristics.

- In [Zivkovic et al., 2011, Zivkovic et al., 2015] a hybrid modelling method is proposed which combines a DSL for managing network devices with the ontology language OWL to provide better specification of domain constraints. The hybrid modelling approach bridges two technical spaces, that of semantic technology and software development, enabling the realisation of various algorithms and mechanisms for software modelling based on application of semantic reasoning. For integrating the metamodels the authors used the metamodel integration approach proposed by [Zivkovic et al., 2007].

- In [Kühn et al., 2011] a methodology for model-driven process and a modelling framework for interoperable, inter-organisational business processes is proposed which relies on the integrated modelling language approach ([Zivkovic et al., 2007]) to combine multiple languages on different levels of the modelling framework.

Table 2.2 classifies the aforementioned approaches based on the integration dimensions of hybrid modelling methods.

Table 2.2: Classification of hybrid modelling methods

Integration dimension	**Hybrid modelling method**
Horizontal	[Shen et al., 2004]: Integration of complementary languages for system decomposition(IDEF0), data(IDEF3) and process flows(DFDs). [de Kinderen et al., 2012]: Extension of ArchiMate with value modelling aspect using e^3value. [Morin et al., 2009]: Extension of domain specific models with feature modelling. [Schroth et al., 2007]: Extension of BPMS with CCTS standard for data modelling. [Zivkovic et al., 2007]: Extension of the BPMN with an organisational aspect.
Vertical	
Cross-wise	[Xu et al., 2010]: Integration of i* method for high-level requirements engineering with low-level technical UML activity diagrams. [Zivkovic et al., 2015]: Integration of low level ontology language OWL for modelling constraints with domain specific language for network device management.
Multi-dimensional (Frameworks)	[Rausch et al., 2011]: Extension of the BPMN with the BPMS method framework. Vertical integration of BPMS business process models used as business level process descriptions with BPMN for detailed technical descriptions. Horizontal integration of BPMN with risks and controls aspects. [Gérard et al., 2011]: MARTE language provides language parts for both vertical refinement of the real-time and embedded systems, as well as for the horizontal partitioning/aspects, such as time, concurrency, software and hardware platforms.

2.2.2 Method Modularisation (Fragments and Chunks)

Following the assembly-based approach for method construction, methods are assembled and integrated out of existing methods components. Method components are basic building blocks, which allow to construct a method in a modular way.

Method Fragments, Method Chunks

Surveying the literature, numerous terms are in use referring to the notion of a reusable method component, out of which the following two are the most prominent: method fragment [Brinkkemper, 1996], [Kühn, 2004], method chunk [Rolland et al., 1999], [Ralyté and Rolland, 2001], [Ralyté, 2004], [Mirbel and Ralyté, 2005]). The term method fragment was made popular by [Brinkkemper, 1996]). Method fragment can be regarded as an atomic element of a method, by analogy with the notion of a software component. According to [Mirbel and Ralyté, 2005] a method chunk is an autonomous and coherent part of a method supporting the realisation of some specific ISD activities. As a part, one chunk is always specified by a descriptor, interface (described as pair of situation and intention) and a body containing the process part and the product part. As pointed out by [Henderson-Sellers et al., 2008], in contrast to fragments being fine-grained atomic method elements, method chunks are more coarse, being a combination of product fragments and process fragments. Hence, method chunk may be built of two or more method fragments. Figure 2.10 summarises the discussion about method components, chunks and fragments by pointing out the relationships and cardinality between the introduced concepts using a metamodel view.

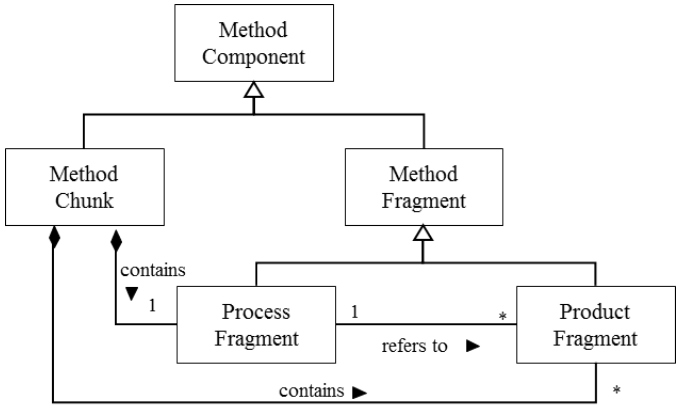


Figure 2.10: Method component metamodel according to Henderson-Sellers et al. [Henderson-Sellers et al., 2008]

Method Fragment Structure

In [Kühn, 2004], the notion of a method fragment is introduced to support the construction of integrated methods based on existing method fragments. A *method fragment* consists of a package and an interface.

Package A *package* is a body of the fragment, containing the constructs of the fragment used to describe one or more method elements (language, mechanisms, process). A package can be atomic or composite. An *atomic package* represents a single, self-contained set of constructs of a single method element type. The package may contain structural elements (first order elements of a method element, which may be exposed via adapters) and content elements (properties of structural elements, not exposable via adapters). A *composite package* contains other atomic and composite packages. Hence, composite package may include subpackages of different method element types (language, mechanisms, process).

Interface An *interface* of a fragment exposes the package content. An interface consists of a fragment specification and adapters. Fragment specification is an informal description of a fragment according to the properties such as name, description, application field, context, classification, change history. An *adapter* is used to expose the inner element of a fragment for integration. One adapter corresponds to exactly one structural element of the package.

Finally, fragments as reusable method assets are stored in the fragment catalogue, categorised according to their characteristics described in the fragment specification. Figure 2.11 illustrates the metamodel of the method fragment. For detailed description as well as usage examples refer to [Kühn, 2004].

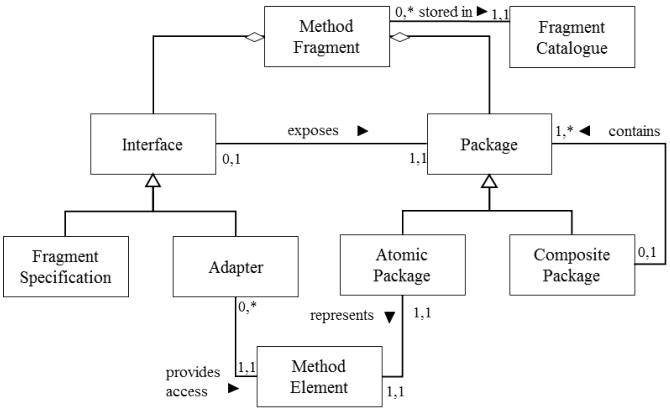


Figure 2.11: Method fragment according to Kühn [Kühn, 2004]

Comparing the notion of the method fragment by Kühn [Kühn, 2004] with other approaches, the following correlations may be drawn. A fragment with atomic content matches to the method fragment definition by Brinkkemper [Brinkkemper, 1996]. A fragment with composite content containing exactly one package describing the method element language and one package describing the process corresponds to the method chunk definition by Ralyté and Rolland [Ralyté and Rolland, 2001].

2.2.3 Method Integration (Patterns, Mappings and Integration Rules)

In the previous section, an overview of existing concepts for method modularisation has been provided. Once method modules exist, their recombination is needed to construct new combined, hybrid methods. In the domain of method engineering only few approaches exist which deal with integration mechanisms for method integration. In [Kühn, 2004] integration patterns are proposed for recurring integration problems. In [Ralyté et al., 2004], high-level generic operators for method engineering have been suggested. In [Zivkovic et al., 2007] a metamodel-based integration approach using mappings and integration rules is suggested. While this may not be a definite list, these three approaches are described in the following.

Patterns-based Approach

In [Kühn, 2004] integration patterns for method integration are proposed based on recurring integration problems. Patterns explicate knowledge about problems and solutions for typical integration problems on a high conceptual level. Kühn proposes a system of interrelated patterns which may be used to guide the method integration decisions. Patterns may be categorised according to the method element they refer to (language, mechanisms, process), integration approach (horizontal, vertical, hybrid/cross-wise) and level-of-coupling (low, intermediate, strong) being the most important criteria. Typical integration pattern for modelling technique is the merge pattern. Figure 2.12 illustrates the merge pattern, as a high level knowledge on merge mechanism for integration. Other typical patterns applicable for modelling language integration exist such as Transformation, Extension, Reference. Some patterns for mechanisms and process integration are Delegation Pattern and Responsibility Pattern, respectively. Detailed descriptions of integration patterns are outlined in [Kühn et al., 2003] and [Kühn, 2004].

Mapping-based Approach

In [Zivkovic et al., 2007] a metamodel-based integration approach using mappings and integration rules for assembly-based method construction has been proposed. The approach extends the enterprise model integration (EMI),

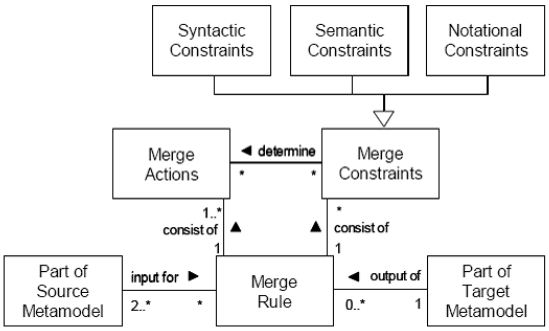


Figure 2.12: Merge Pattern according to [Kühn et al., 2003]

an approach based on metamodel integration patterns [Kühn et al., 2003], by introducing the notion of mappings and integration rules. The approach suggests a mapping-based metamodel integration method (a meta-method) consisting of a metamodel mapping language, integration rules as mechanisms, and a model-based, step-wise integration procedure. Based on the metamodel heterogeneity problems, the authors identify the set of corresponding mappings as problem solutions, which are used to describe which metamodel elements should be integrated and how. The integration rules represent a dynamic part of the method, which for given source metamodels produce a new target hybrid metamodel based on mappings. Furthermore, a generic set of integration rules applicable for various mapping types is suggested. The approach relies on the integration principles in terms of integration patterns such as Transformation, Merge, Reference advised in the EMI approach. In the following, some details of this approach are provided.

Metamodel Heterogeneity Heterogeneity problems may arise when integrating vertically and/or horizontally different metamodels. Due to the fact that metamodels represent structural data information, for the categorisation of heterogeneity problems, the authors refer to the fundamental information heterogeneity known from the field of database schema integration ([Batini et al., 1986], [Ouksel and Sheth, 1999]). Hence, no matter what kind of integration orientation is considered, syntactical, structural and semantic metamodel heterogeneity may occur.

- *Syntactical metamodel heterogeneity.* Syntactical heterogeneity describes the difference in formats, in which metamodels are serialised. Metamodelling platforms may serialise metamodels based on different formats (database or file based, different proprietary schemes etc). One approach to deal with syntactical (serialisation) heterogeneity is to agree on exchange based on standard paradigms and formats (e.g XMI).

- *Structural metamodel heterogeneity.* Structural heterogeneity can be devided in representational and schematic heterogeneity. *Representational heterogeneity* is about differences in meta-metamodels used, that influences the availability of metamodelling primitives (classes, attributes, supported relationship types, single and multiple inheritance, etc.). *Schematic heterogeneity* occurs when the same semantic concepts are described in metamodels in a different way using different metamodelling constructs. For example, a concept in a metamodel may be defined using a single class (Student) or as a hierarchy of classes (Student, Person).

- *Semantic metamodel heterogeneity.* Finally, differences in the intended meaning of metamodel elements are subsumed by the semantic heterogeneity. Applying the categorisation of ontology-level mismatches [Klein, 2001, Noy, 2004] for metamodels, elements in metamodels may be equivalent, related or non-related (orthogonal).

The nature of metamodel heterogeneity implies a step-wise solution approach. First, syntactical conflicts, in terms of different serialisation formats, need to be solved. This level of heterogeneity occurs when integrating metamodels between different tools/paradigms. Once there is a consensus about the common serialisation format, structural heterogeneity may be tackled. At this level, representational conflicts in terms of different meta-formalisms must be aligned first before dealing with structural integration problems. Assuming that syntactical and structural representation heterogeneity issues have been resolved, under the premise of having one meta-metamodel, the focus of the approach is on solving conceptual conflicts between metamodels, i.e. schematic and semantic heterogeneity.

Metamodel Mappings The cornerstone of the metamodel integration approach is the mapping formalism. The basic notation of the mapping language is a metamodel mapping. According to [Zivkovic et al., 2007], a *metamodel mapping* is used to capture both the structural and the semantic relation between concepts of two metamodels. Based on the structural and semantic relations between elements, mappings such as equivalence, relation, non-relation may be differentiated. *Equivalence mapping* states the semantic equality of the elements. *Related mapping* may represent *generalisation*, *aggregation*, *composition*, *association* or *classification* of metamodel elements. *Non-relation mapping* denotes simply orthogonal metamodel elements. Mappings may be further classified based on their structural dimension: 1) mapping type - concerns the type of metamodel elements on both sides of the mapping (A-attribute, R-relation, C-class, e.g. A2C, C2C, etc.). 2) mapping cardinality - the number of connected elements. For example, the attribute *Performers* may be mapped to the class *Processor* using

the related mapping variant *attribute-2-class (A2C)* having cardinality *1-1*. Figure 2.13 illustrates the metamodel of metamodel mappings.

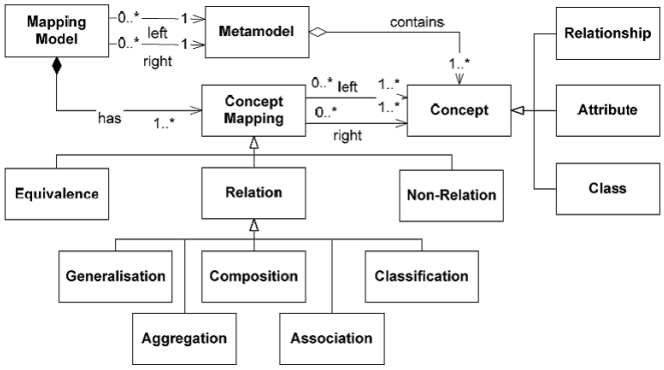


Figure 2.13: Metamodel of metamodel mappings according to [Zivkovic et al., 2007]

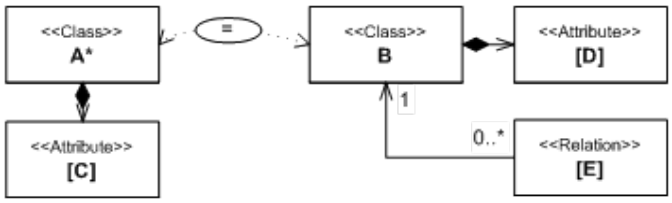
Integration Rules While the mappings represent the structural/static part, integration rules are considered as a dynamic part of the metamodel integration method. Hence, the integration rules represent a set of algorithms for metamodel integration. A rule takes as an input a specific mapping, a set of metamodel elements the mapping connects, and produces as an output, a target metamodel element. Based on the specific metamodel mapping variant, a rule provides an integration execution algorithm. In the previous example of the related mapping, we may want to apply an integration rule which creates a relationship between the container class of the attribute *Performers* and the class *Processor* called *RPerformers*. Based on a defined set of mapping types, one may also identify a set of generic integration rules, which apply for specific mapping types, such as *mergeC2C* rule, *generalizeC2C* rule etc. Table 2.3 illustrates an integration rule specification for the rule MergeC2C. The MergeC2C rule specifies how two metamodel elements of type class (C2C) are to be integrated. A more detailed list of rules is given in [Zivkovic et al., 2007].

The distinction of different mapping types as well as of typical integration rules is useful, since it allows their generic implementation within the metamodelling platform, as a set of routines one can choose from for specific integration scenario. Table 2.4 gives an overview of integration rules categorised according to the mapping variability criteria (semantic relations and structural variations). Note that not all possible combinations are listed. For a detailed list of integration rules refer to [Zivkovic, 2006].

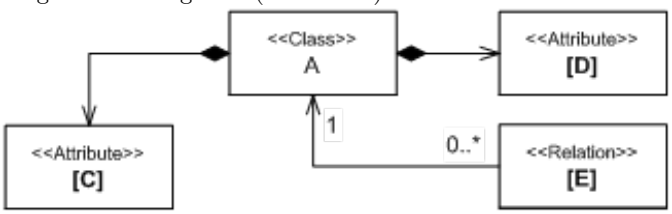
Table 2.3: Specification of the integration rule MergeC2C [Zivkovic et al., 2007]

Rule name:	**Merge C2C**	
Description:	Elements are semantically equivalent. To avoid redudancy of concepts in the integrated metamodel, the equivalent elements are merged into one element in the target metamodel.	
Mapping details (Rule condition):		Rule action:
Semantics	Equivalence	1. One of the affected classes is
Structure	C2C	declared as primary.
Meta-metamodel constraints	n/a	2. The primary class is a leading
Left/Right	A*, [C] / B, [D], [E]	element in the merging algorithm. 3. All attributes and relationships
Target Concepts	A, [C], [D], [E]	are transfered from the overlapping classes to the primary class.

Source metamodel fragments (rule input): *primary concept, [ ] connected concept



Integrated target model fragment (rule result):



Change-centric Approach (Generic Operators)

In [Ralyté et al., 2004], Ralyté et al. introduce a topology of high-level generic operators for change-centric method engineering. The operators are generic in a way that they may be instantiated for any of the method engineering approaches (ad-hoc, paradigm-based, extension-based assembly-based, refer to Section 2.1.6). While providing a metamodel of a method, they identify the basic elements of methods for which generic operators may be defined. Furthermore, the specialisation of operators is done for the product part and for the process part, based on their metamodels. For example, a typical operator for the integration-based assembly of product models would be *ConnectViaMergeClass*, whereas for process models *ConnectVieMergeAc-*

Table 2.4: Classification of mappings and integration rules based on [Zivkovic et al., 2007]

	Equivalence	**Generalisation**	**Related Aggregation**	**Association**
C2C	merge() map() abstract()	generalize()	aggregate()	associate()
A2A	map()			
A2C	merge() map()	generalize()	aggregate()	associate()
R2R	merge() map()			
R2C	merge() map()		aggregate()	
A2R	merge()		aggregate()	

C - class, A - attribute, R - relation

tivities. The authors also classify operators according to the type of changes they generate: naming changes, element changes (changing element properties) and structural changes. The latter category is important, since it refers to the changes on a set of model elements. These can be further divided into inner changes (within one model) and inter-model changes. Figure 2.14 provides the list of generic operators according to [Ralyté et al., 2004].

2.2.4 Life Cycle of Hybrid Modelling Methods

Hybrid modelling methods are created based on existing method fragments amalgamating different modelling approaches into one, new hybrid modelling approach. Similar to other products in engineering, methods do have a life cycle in which they are created, used and finally retired. For example, in software engineering, according to UP [Jacobson et al., 1999], software products and projects undergo four main phases in their life cycle such as inception, elaboration, construction and transition. In [Kühn, 2004], Kühn devides the life cycle of a method into four main phases: 1) conception, 2) implementation 3) introduction 4) operation. Each phase consists of specific tasks, roles and actors. The method life cycle is characterised by an iterative-incremental execution of its phases, thus enabling early feedback loops and parallel execution of its activities. Figure 2.15 illustrates the life cycle. In the following, an overview of the phases is given. A detailed description of each of the phases and their activities is given in the mentioned literature reference.

Conception

The conception phase starts with the method *goal definition*. Typical decisions that are made here are regarding the application scenarios and type of users of the method, the expected results of the method as well as the

Object	Operator	Description
Element	*Rename*	Change the name of an element.
	Add	Add a new element in the model.
	Remove	Remove an element from the model.
	Merge	Two separate elements become one element.
	Split	An element is decomposed into two elements.
	Replace	An element is replaced by a different one.
	Generalize	An element is created as a generalization of two elements.
	Specialize	Specialise an element into two sub-elements.
Compound	AddComponent	Add a component into an element.
	RemoveComponent	Remove a component from a compound element.
	MoveComponent	A component is repositioned in the structure of a compound element.
Property	*Give*	Add a property to an element.
	Withdraw	Remove a property from an element.
	Modify	Change a property in an element.
	Retype	Change the type of an element.
Model Element	*Instantiate*	Instantiate an As-Is model element into To-Be model element.
	Abstract	Create a To-Be model element as an abstraction of an As-Is model element.
	ConnectVia Specialization	Define an element from one model as a specialization of an element from another model. An is-a link is created between these two elements.
	ConnectVia Generalization	Generalize two elements from different As-Is models into a super-element in the To-Be model.
	ConnectVia Composition	Create a compound element in the To-Be model containing as components elements from two different As-Is models.
	ConnectVia Decomposition	Define an element as a component of an element from another model.
	ConnectVia Association	Add an association link in the To-Be model between two elements from different As-Is models.
	ConnectViaMerge	Two similar elements from different As-Is models become one element in the To-Be model.

Figure 2.14: Method engineering generic operators according to [Ralyté et al., 2004]

chosen methodology for method use. Clearly defined and structured goals become input for the *requirements specification*. In this step, key elements of the method are analysed regarding needed features. For the modelling language part, it is important to define which aspects/views should the language support, and based on it, which classes, relations and attributes are part of those aspects. Similarly, it must be specified which algorithms and mechanisms are required and which life cycle model will be used. *Consolidation* has the purpose to evaluate and prioritise specified requirements with respect to the achievement of previously defined goals. Finally, in the *specification* the method is conceptualised according to the consolidated set of requirements. The conceptualisation is done in detail, such that represents a valid input for the implementation phase.

Implementation

The implementation phase starts with the *selection* of the corresponding existing method fragments. This is done according to the previously defined goals, requirements and method specification. Method fragments are stored in a method fragment repository. The result of the selection is the

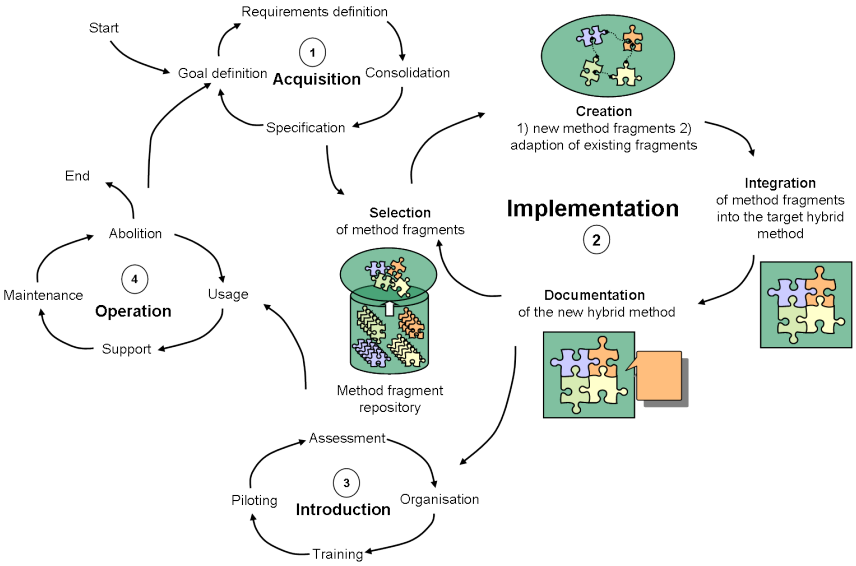


Figure 2.15: Life cycle of hybrid modelling methods (adapted based on [Kühn, 2004] and [Kühn et al., 2011]

set of target fragment candidates to be integrated into a hybrid method. The next step in the implementation phase is the *creation* of the method fragments. In this step, the creation of the new fragments and/or adaptation of the existing fragments are performed. Once all needed fragments are selected, created and/or adapted according to the specification, the method *integration* may be performed. Here, different types of fragments such as language, algorithms and mechanisms and process parts are integrated to build a hybrid method which is then operationalised in a specific method engineering environment such as a metamodelling platform. In the last step of the implementation phase, the *documentation* of the new hybrid method is created.

Introduction

The introduction phase of a modelling method deals with an initial deployment and usage of the method within a company. Introduction starts with a *organisation* step, in which organisational aspects such as change management, involvement of key persons, etc., as well as technical measures such as hardware and software infrastructure, etc., for a successful installation of the method within a company are done. *Training* is another important step in the introduction, in which, involved persons are trained to use the method as well as the corresponding modelling tool. The key task within the introduction is the *piloting* phase. During method piloting, the introduced method is used under real-world conditions within a company, however, for

a limited group of dedicated problem domains. The purpose of piloting is to evaluate the modelling solution under realistic conditions. The results of piloting are considered in the *assessment* phase. Here, the method itself as well as the modelling tool is given a valuable feedback regarding the suitability for the problem-domain.

Operation

The operation phase takes place after successful introduction, in which the method is *used* company-wide for all relevant problem domains. From the life cycle perspective of the method this is a very valuable phase where the method is extensively used and valuable feedback and ideas for further advancements are gathered. Parallel to *method usage* phase, *support* activities take place. Support can either be domain-related or of a technical nature. In the *maintenance*, feedback gathered during piloting and usage is evaluated and based on it, the method is extended or adapted to problem-specific needs. These are usually small adaptations in comparison to the new version of the method which triggers the whole new life cycle of the method. The *retirement* of a method takes place if a method doesn't satisfy the requirements of the problem domain anymore. In that case, a method is completely removed or replaced by a new method version. In the retirement, it is important to collect the feedback and analyse the causes for the abandonment, in order to take improvement measures for the next method version.

2.3 Case Studies in Hybrid Modelling Methods

In the previous sections, the concept of hybrid modelling languages and integrated methods as well as the various approaches to integrate methods and particular languages have been introduced. In the following, three case studies from research and industry are outlined in order to exemplify the necessity and practicability of hybrid methods and modelling. The case studies feature in particular the mapping-based integration approach to metamodel integration introduced in previous section.

2.3.1 Integration of BPMN and Organisational Modelling

The underlying case study of the integration of the Business Process Model and Notation (BPMN) [OMG, 2013] and ADONIS Business Process Management Systems (BPMS) method [BOC, 2015] has been initially used to validate the mapping-based metamodel integration approach described in [Zivkovic et al., 2007]. This hybrid modelling method has meanwhile become a standard method of the business process management tool ADONIS and is used widely in industry [Murzek et al., 2013].

In the last decade, BPMN gained significant attention in business process management community, both in industry and research, and became a de facto standard for business process modelling on both business and IT level. However, BPMN lacks advanced concepts for the modelling of organisational aspects of processes. For example, capacity-based process model simulation for resource planning is possible only if appropriate organisational concepts (roles, workers, organisational units and alike) are available.

According to [Zivkovic et al., 2007], a metamodel integration project begins by selecting source metamodels to integrate, which is followed by the identification of mappings and selection of appropriate integration rules. Stereotyped UML class diagrams are used to represent source metamodels, in order to distinguish types of metamodel elements such as classes, relationships and attributes. Mappings are captured using a specific mapping notation. Source metamodels annotated with mappings represent the mapping model. Figure 2.16 illustrates mapped metamodels with identified integration points.

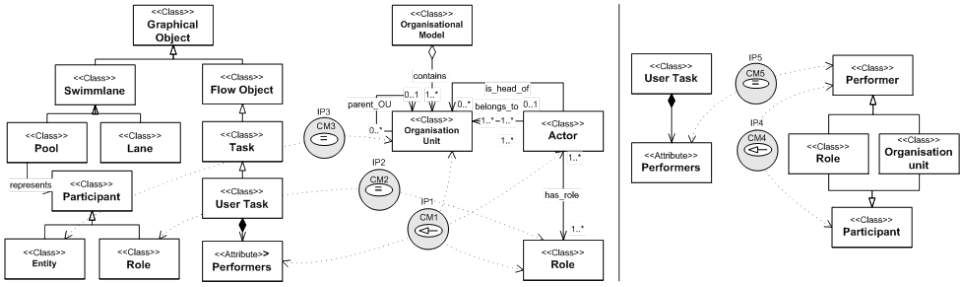


Figure 2.16: a) Source metamodels annotated with mappings and integration points, b) Revisited mappings [Zivkovic et al., 2007]

In the presented example, several integration points (IP1, IP2, IP3) and mappings (CM1, CM2, CM3) have been identified. Accordingly, for each mapping, an appropriate integration rule is applied.

- *IP1/CM1.* The rule *GeneralizeA2C* transforms attribute *Performer* in a super class of classes *Organisation unit*, *Actor* and *Role*.
- *IP2/CM2.* The rule *MergeC2C* integrates equivalent classes *Role* from both source metamodels.
- *IP3/CM3.* The rule *MergeC2C* merges classes *Entity* and *Organisation unit*, where the latter is chosen as the primary class.

During the integration step, the rules are applied and the concepts mapped are transformed into a new integrated metamodel. The remaining unmapped concepts are copied to the integrated metamodel. Since the result of initial integration may result in a metamodel that doesn't conform

to the meta-metamodel, the integration step is repeated until all conflicts are resolved. In the introduced example, with the new super class *Performer*, the classes *Organisation unit* and *Role* now have two parent classes, which implies multiple inheritance. Given the fact that the underlying meta-metamodel does not support multiple inheritance, this inconsistency must be resolved. Hence, we return to the mapping phase and define two new mappings CM4 and CM5 based on integration points IP4 and IP5. Figure 2.16 b) illustrates an excerpt of the the intermediate integrated metamodel. Based on the CM4 mapping, the class *Participant* becomes a child of the class *Performer*, and consequently a parent of the classes *Role* and *Organisation unit*. These two classes are at the same time children of the *Performer* class, such that this class structure implies the use of the complex *EmbedC2C* rule [Zivkovic, 2006]. This way, the class *Participant* is embed between *Performer* and its children classes. The *MergeA2C* rule is selected and applied for the mapping CM5, having the class *Performer* marked as primary. In doing so, an aggregation relationship *performers* from the class *UserTask* to the class *Performer* is created, in order to retain the semantics of the former attribute *Performers*. Figure 2.17 illustrates the final state of the new integrated metamodel.

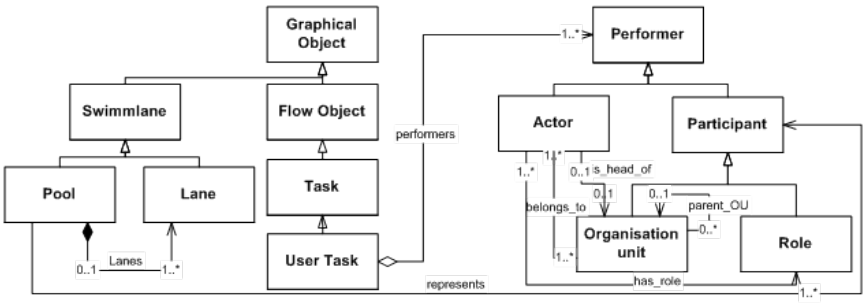


Figure 2.17: Integrated metamodel that extends BPMN organisation modelling concepts [Zivkovic et al., 2007]

The benefit of the new hybrid language as an extension of the BPMN is twofold. It now supports modelling of BPMN pools and connecting those pools with organisational participants which may be either organisational units or roles. Furthermore, it allows for assigning performers, which may be actors, units or roles, directly from the organisational structure to user tasks in business process models.

2.3.2 Model-driven Development of Interoperable, Interorganisational Business Processes

The following case study of method integration considers cross-wise integration approach to build up a holistic modelling framework and a development

process for model-driven development of interoperable, inter-organisational business processes according to [Kühn et al., 2011]. The core of the framework builds up a common modelling language that integrates all participating metamodels into one hybrid metamodel used by all involved actors. The metamodel integration has been conducted again based on the mapping-based approach described in [Zivkovic et al., 2007].

Table 2.5: Variety of languages in the modelling framework for interoperable, inter-organisational business processes according to [Kühn et al., 2011]

Phase	**Step/Objective**	**Actor**	**Language**	**Result**
Development	Transform business process models to BPMN models Refine BPMN models Workflow and document description	IT architects Application developers	BPMN Business information model CCTS ER XPDL	BPMN models Document description Integrated process and data models
Execution	Transform BPMN models to BPEL/WSDL Enrich, test, deploy, and execute business process	Application developers IT operators	BPEL WSDL XPDL XML XSD	Executable business process definitions (BPEL) Executable business documents

According to [Kühn et al., 2011], in order to achieve interoperable business processes, different organisational levels, such as strategic, operational, development and execution level are to be considered during the analysis of the business processes and of the subsequent development steps related to the implementation of the supporting information systems. On a strategic level, interoperability has to consider governance as well as issues of strategic alignment. To achieve interoperability on operational level, the business processes of the two organisations have to be aligned. This includes agreement on the properties of exchanged products and the quality levels of exchanged services. Additionally, the interaction points of the business processes, the related business rules and the exchanged business documents have to be specified. On a development level, business processes are implemented either using organisational techniques or using software technology. On this level, business processes are enriched with either organisational and/or technical data. Execution level considers the actual execution of processes. On each of those levels different modelling languages or data formats are used within the particular development phase to capture various kinds of data. This includes business processes, their interactions, the exchanged business

documents and business information, the requirements for the underlying information systems, the design of the business data, the design of executable workflows, and necessary aspects of the underlying technical infrastructure. To illustrate the variety of languages on different interoperability levels and for different development phases, Table 2.5 lists steps, actors, languages and results on the development and operational level. A complete overview may be found in [Kühn et al., 2011].

The metamodel integration process can be used to create either common metamodels or to map existing metamodels to be used in conjunction within the metamodel framework. These metamodels are used as a common schema to share all necessary information from strategic information down to information relevant for the process execution. Metamodels are integrated horizontally, i.e. different aspects are merged. For example, in the development phase, BPMN process models need to be interlinked with the data models, in order to capture the input and output documents of certain process tasks. Furthermore, metamodels are mapped vertically between levels, to enable seamless transformation (e.g. from operational to development level).

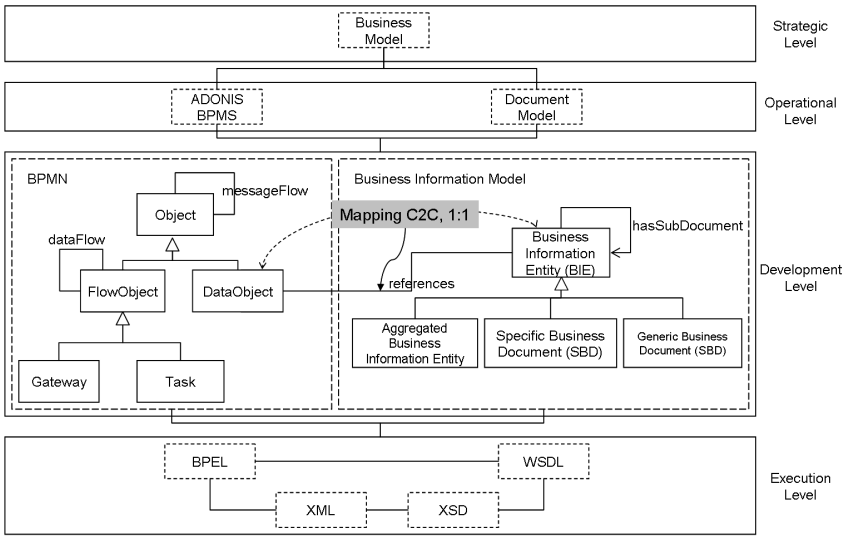


Figure 2.18: Overview of the integrated metamodel (Modelling framework for interoperable, inter-organisational business processes [Kühn et al., 2011]

Particularly, considering the project specific requirements for the modelling language from the case study, the resulted modelling language represents a set of metamodels which are distributed on different layers according to phases of the model-driven development approach. Figure 2.18 depicts the simplified integrated metamodel. In particular, the illustration outlines the integration of the languages BPMN and CCTS-based Business Informa-

tion Model (BIM) on the development level. Using the integration approach with mappings and integration rules, an associative metamodel mapping of type C2C and cardinality 1-1 between the BPMN metamodel element *DataObject* and the BIM metamodel element *Business Information Entity (BIE)* has been identified. Consequently, the integration rule *associateC2C* has been applied which resulted in the creation of the relationship *references* between the DataObject and BIE. The benefit of the integrated language is the ability to model BPMN processes and anotate them with data objects which are described in detail using the CCTS-based BIM standard.

2.3.3 Hybrid Modelling Method for Consistent Physical Devices Management

As part of the case study within the project MOST[2], a hybrid modelling method for the modelling of physical devices has been designed. The goal of the case study was to explore the application of semantic technologies in software and system modelling.

The PDDSL method introduces a hybrid modelling approach. A domain-specific modelling language is used to capture the structure of physical devices. On the other side, the ontology language OWL2 is applied to specify semantic constraints on top of physical device models. Furthermore, PDDSL mechanisms provide consistency checking of physical device models whereas PDDSL modelling procedure guides users through the network configuration process.

In [Zivkovic et al., 2011, Zivkovic et al., 2015] the design and implementation decisions for all parts of the method have been described in detail. In the following, after the case study is motivated, the focus is set on the design of the modelling language part of the hybrid method and, in particular, on the metamodel integration decisions.

Case Study

According to [Zivkovic et al., 2015], one of the challenges in the management of physical device equipment is its consistent configuration. Maintaining large number of devices of various types may be a complex, error-prone and cost-intensive endeviour if it is done without an adequate tool support. Most of the state-of-the-art tools in this sector fail at providing a consistent support when it comes to guiding the users in complex device configuration scenarios.

[2]MOST was an EU research project that pursued the goal of marrying ontology and software technology. In particular, the focus was set on the integration of ontology languages and reasoning technology and modelling languages. By the time of the project end, MOST offered, to project members best knowledge, the first systematic approach of bridging the model and ontology technical spaces [Staab et al., 2010], [Pan et al., 2013].

It is acknowledged that modelling methods allow for systematic capturing of relevant domain knowledge in terms of models. Hence, the complexity of network device configurations may be significantly reduced by capturing the semantics of device types and their configurations in domain-specific models and using domain-specific modelling tools. It is known that formal ontology languages such as OWL2 allow for defining precise constraints of models through formal semantics. Domain-specific models of physical devices that are annotated with semantic constraints may be transformed to an ontology, in order to enable the application of standard reasoning mechanisms such as consistency checking, subsumption checking, classification, explanation and justification.

When modelling physical devices, two aspects appear to be relevant. On the one side, modelling of physical device types defines the structure of single physical devices (product name, cards, slots, number and type of slots, etc.). This structure may also introduce certain constraints on valid configurations of that particular device. On the other side, modelling of physical device configurations/instances is another aspect, in which specific configuration of a network device is captured. To illustrate the modelling scenario, Figure 2.19 displays the structure of the device type *Cisco 7603* (on the left) and its specific configuration (on the right).

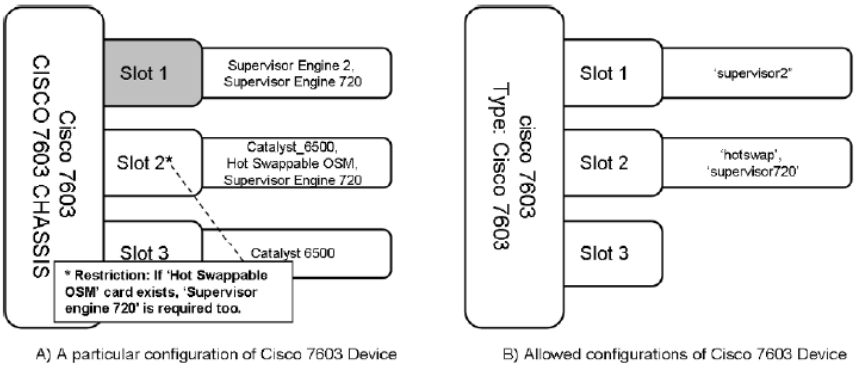


Figure 2.19: Specification and sample configuration of network device Cisco 7603

Language Design

Capturing of physical devices in terms of models as introduced in the case study implies the availability of two major concepts in the language design 1) support for both linguistic and ontological instantiation 2) support for semantically-rich constraints definition. Concerning the first concept, both network device types and concrete device instances have to be modelled at the same modelling level. Hence, PDDSL is designed according to the two-dimensional metamodelling architecture [Atkinson and Kühne, 2003]. The

second language concept refers to the fact that the language has to support capturing of complex constraints on device type structures, which, in turn, are validated against concrete device instantiations. Therefore, PDDSL is integrated with OWL2 [Motik et al., 2009b]. The abstract syntax of the hybrid language consists of metamodels of PDDSL and OWL2, both integrated using well-defined integration points. Figure 2.20 provides an overview of the particular metamodel fragments as well as of their integration points.

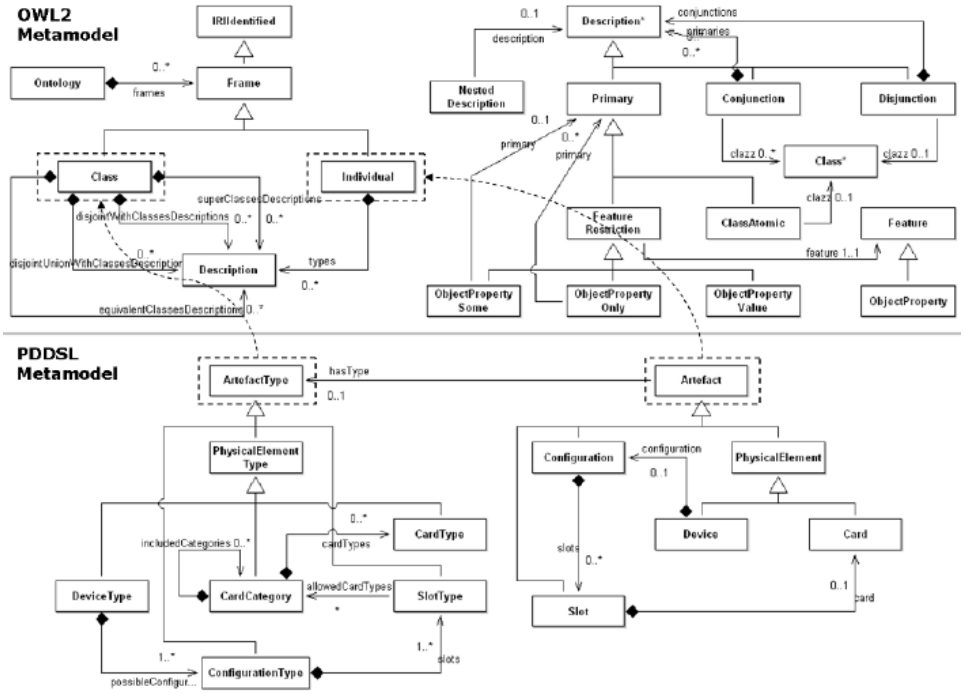


Figure 2.20: Overview of the metamodel parts and integration points [Zivkovic et al., 2011]

The PDDSL Metamodel

The PDDSL metamodel part consists of constructs for the modelling of device types and device instances. In order to enable ontological instantiation, the relationship *hasType* connects the class *Artefact* and the class *ArtefactType*, with instantiation semantics, such that artefact instances are based on their artefact types. For example, the card *supervisor2* is an inventory instance of the card type *Supervisor Engine 2*, both defined on the same linguistic level.

The OWL2 Metamodel

The OWL2 metamodel is constructed based on abstract syntax elements defined in OWL2 Manchester Syntax [Horridge and Patel-Schneider, 2009]. The syntax is object-centred and frame-based, unlike other OWL2 syntaxes which are axiom-based. The upper part of the Figure 2.20 illustrates a small, but relevant subset of a rather complex OWL2 metamodel. *Classes* and *Individuals* are defined as ontology *Frames*. Details of classes are defined using *Descriptions*. Descriptions allow for simple or complex class expressions based logical operators, or existential and universal quantification operators. In OWL, such class descriptions can be used as instance types through *types* reference.

Metamodel Integration

Metamodel integration is a complex language engineering task. The complexity arises from the fact that the integration should ideally produce a hybrid metamodel which preserves both syntax and semantics of the single metamodels. The preservation of syntax and semantics is of crucial importance, in order to retain the compatibility with mechanisms already built for single languages. For example, a hybrid language that integrates OWL2 must not change semantics of OWL2 during integration, due to the fact that any change may cause reasoning mechanisms not to work anymore. Therefore, metamodel integration subsumes deep understanding of syntax and semantics of the integrating metamodels, in order to identify non-invasive or at least semi-invasive metamodel integration points. We refer to the mappings for bridging structural languages and ontology languages [Walter and Ebert, 2009] when integrating PDDSL and OWL2 metamodels. In doing so, we apply metamodel integration rules defined in [Zivkovic et al., 2007]. Although the integration appears to be trivial, it is at the same time powerful, considering the outcome. The PDDSL metamodel element *ArtefactType* is said to be the subclass of OWL2 *Class*, thus inheriting the rich OWL2 class expressiveness. Similar is done between *Artefact* and *Individual*. This way, PDDSL artefact types and artefacts can take part in OWL-based constraint definitions, which was our major goal for the integration. Obviously, the integration is semi-invasive (invasive from the viewpoint of PDDSL and non-invasive from the viewpoint of OWL). Being subtype of OWL classes, PDDSL classes inherit new attributes of its supertypes. However, since the nature of the conducted changes is extensional (new attributes added), the changes haven't influenced the PDDSL semantics. In addition, the syntax and semantics of OWL2 remained intact in the hybrid language, allowing the application of reasoning mechanisms.

Language Semantics

The semantics of the hybrid PDDSL language consists of several parts. Given the fact that PDDSL constructs inherit from OWL constructs, the PDDSL core semantics and the connection to OWL2 are explicitly defined using OWL2 and transitively using Description Logics (DL) (OWL2 is formally defined using DL [Motik et al., 2009a]). The operational semantics are implemented as model transformations that translate hybrid PDDSL models to pure OWL2 ontologies, considering the open-world (OWA) and close-world assumptions (CWA) for proper reasoning [Miksa et al., 2010].

Benefits of Hybrid Modelling

Instead of defining a single, monolithic language from scratch, we were able to combine two domain specific languages for modelling device types and device instances and one state-of-the-art ontology language for modelling semantic constraints, and profit from the mechanisms built on top of those languages. Furthermore, hybrid languages facilitate clear separation of concerns in the modelling solution with clearly defined integration points, both on the language engineering level as well as on the modelling level. In the latter case, our hybrid language thus clearly differentiates between two modelling roles: Device experts and Device users. Device experts devise the device type ontology, specific device types and their semantic constraints. Device users have the role to configure device instances following the semantic constraints defined by device experts. The hybrid concrete syntax visualisation contributes to better user experience by providing role-specific model perspectives.

2.4 Chapter Summary

In this chapter, we introduced the basic concepts related to the existing work on the engineering of modelling methods in general and of hybrid modelling methods in particular. Special focus was set on the modularisation and integration approaches. In Section 2.1 we introduced the basic notions of a modelling method and elaborated in detail on the extended modelling method framework and its elements. In doing so, we also contributed with the additional categorisation of modelling method hybrid mechanisms into configurable and adaptable in Section 2.1.4. In the same section, an additional classification of modelling method mechanisms based on application purpose has been suggested. In Section 2.2, we elaborated on method integration approaches for hybrid modelling methods. We first revised the integration dimensions and added a fourth composite, multi-dimensional category (see Section 2.2.1). After that, an overview of existing conceptual approaches for method modularisation and integration has been provided.

Finally, in Section 2.3 we underpinned the introduced concepts by a selection of case studies in hybrid modelling method engineering.

The introduced approaches for assembly-based modelling method integration contribute with high-level conceptual solutions for designing hybrid modelling methods. However, a sound formalism for modular modelling method engineering, and in particular, for modular metamodel engineering on the technical level hasn't been addressed. Closing this gap is the focus of the underlying work.

Chapter 3

Concepts for Modelling Language Engineering

"Anything you can do, I can do Meta."

Daniel Dennett

From the method engineering point of view, modelling language engineering is a special discipline of method engineering, which concentrates on product part of the method, the modelling language. Language engineering in general is a special field of software development that focuses on creation of any kind of software languages (programming, domain-specific, textual, visual, etc.). In its simplest form, a common sense on the language anatomy is that a language usually consists of the syntax (abstract syntax), semantics and notation (concrete syntax). There are various approaches to define core language elements such as graph-based or metamodel-based approaches. It has been recognised that metamodelling, a formalism based on metamodels, is the most practical choice to specify languages, i.e. their abstract syntax. In the underlying work, the focus is set on modelling languages, and in particular, on their metamodel-based definition. Nevertheless, the goal of this chapter is to provide a broader overview of the language engineering field in general, with the special focus on analysing the existing work on metamodel-based modelling language definition.

The chapter is structured as follows. In Section 3.1 we explain the engineering phenomenon of language layers, a technique to structure the language and model[1] concepts and data, common to different language engineering approaches. In Section 3.2, we elaborate on modelling language anatomy, i.e. on the basic language elements. In Section 3.3 a thorough

[1]Model is simply a product of using a language, may be a text, a graphical model, a program code, etc.

overview of approaches for language definition is provided. Here, approaches for the definition of abstract syntax, concrete syntax and semantics are separately introduced. Finally, in Section 3.4, we discuss existing approaches to metamodel-based language definition. In doing so, we first define the capabilities of metamodelling languages, which serve as an evaluation framework to analyse and compare existing metamodelling languages. Section 3.5 summarises the chapter.

3.1 Language Engineering Layers

In the field of language engineering (modelling and programming languages), the information about systems described in terms of models or code is structured into several layers. The lowest layer is on the level of the universe of discourse (system, real-world data), layer (L0). On top of it, models are used to represent the system (L1). This layer is usually called the application layer or the model level. Models and code are described using languages, which reside on the next layer (L2). This layer is usually called tool level or the language level. Similarly, the next level consists of a meta-language for describing languages (L3). Usually, the meta-language itself is described on the same level, by self-definition (or bootstrapping), since it already contains very basic constructs which do not require any further levelling. This kind of levelling is called *linguistic instantiation*[2], and is widely adopted architecture in the conventional programming language design [Aho et al., 1986], information systems development (IRDS, ANSI-X3.138-1988 [Parker, 1992]), CASE tool design (CDIF, [Chen, 1993]), graph technology [Ebert and Franzke, 1995] modelling and method engineering community ([Karagiannis and Kühn, 2002], [Frank, 2002]) and most-recently in the field of model-driven engineering (MDA, [Mukerji and Miller, 2003], [Kleppe et al., 2003]). However, this kind of levelling is merely a practical agreement. Other approaches such as the *ontological instantiation* allow for having instances and types on the same linguistic level [Atkinson and Kühne, 2003]. In the field of knowledge representation and semantic technologies, ontologies are structured in two levels, the type level, s.c. ontology *Tbox* and the instance level, s.c. ontology *Abox*. Table 3.1 summarises the terminology used in multiple computer science fields that follow the layered approach to structure information.

[2]The term instantiation is however not precise, since the system is *represented* by models, and models, languages *conform* to languages, meta-languages. This kind of levelling/layering is thoroughly discussed in the field of MDE in [Bézivin and Gerbé, 2001]

Table 3.1: Terminology of language layers in different computer science fields

	Layer 3	**Layer 2**	**Layer 1**	**Layer 0**
Programming language design	metasyntax, meta-language	grammar, language	program code	run-time data
Model-based and model driven development	meta-metamodel, metamodelling language	metamodel, modelling language	model	run-time data, system, universe of discourse
Graph theory	metaschema	schema	graph	run-time data, system, universe of discourse
Knowledge representation	Ontology Tbox	Ontology Abox/Ontology Tbox	Ontology Abox	universe of discourse

3.2 Modelling Language Anatomy

There are different works discussing the anatomy of modelling languages. According to [Karagiannis and Kühn, 2002] a modelling language consists of a syntax, semantics, notation. Syntax, also known as abstract syntax, describes the structure of the language, whereas, the semantics provide a meaning to the abstract syntax elements. Notation, also referred to as concrete syntax, defines the representation of the abstract syntax elements (e.g. graphical symbols, text). In addition to the core elements of the language, Greenfield [Greenfield et al., 2004] introduces another form of syntax, the serialisation syntax. While concrete syntax is human-readable, serialisation syntax may not be human-usable and is used to persist and interchange language expressions in a serialised form. Kleppe [Kleppe, 2009] generalises the serialisation syntax as being a form of a concrete syntax. Hence, according to Kleppe, modelling language has one abstract syntax and one or more concrete syntaxes. Selic [Selic, 2011] also points out that the abstract syntax is a pivotal element, for which multiple concrete syntaxes may exist, and on the other side, exactly one semantics. A new aspect being brought by Kleppe to the language concepts, and which has been underlined by Selic is the optional existence of language interfaces, as an ability of languages to interwork with other languages.

Figure 3.1 illustrates the anatomy of modelling languages. Single elements of the modelling language are discussed in the subsequent subsections.

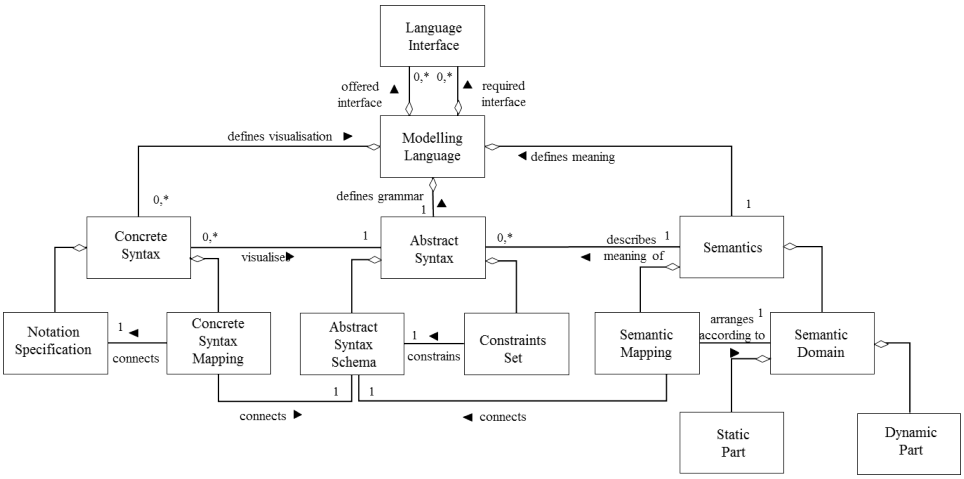


Figure 3.1: Modelling language anatomy

3.2.1 Abstract Syntax

Abstract syntax defines the main concepts of the language and relationships between them. Abstract syntax[3] is regarded as a fundamental, pivotal element of the modelling language [Greenfield et al., 2004], [Kleppe, 2009], [Selic, 2011]. It is a central element connecting together multiple concrete syntaxes as well as the language semantics[4]. As pointed out in [Selic, 2007], besides main abstract syntax concepts, the abstract syntax also consists of composition rules that represent a kind of algebra on how to combine the concepts to produce valid models. Composition rules take form of constraints and well-formedness rules. In [Harel and Rumpe, 2000] these are called context conditions, which further restrict the abstract syntax[5].

[3]As Kleppe notices, the term "abstract syntax" comes from natural-language research, where it means the hidden underlying, unifying structure of a number of sentences ([Kleppe, 2009], p.75)

[4]Traditional approaches for textual language design - compiler technology - centred the focus of the language on the human-oriented concrete syntax and the machine-interpretable abstract syntax tree, thus ending up with having a single concrete syntax representing the language. Recognising the changing nature of software languages [Kleppe, 2009], modern language theory shifts the focus on the abstract syntax, thus allowing the language to have multiple i.e. arbitrary concrete syntaxes all of them mapped to one common representation, the abstract syntax.

[5]Some consider constraints as part of semantics. As stated in [Harel and Rumpe, 2004]... in compiler theory, constraints on the fourth layer are often called semantic conditions because semantic considerations trigger them. However, the constraints affect only the syntax; they do not contribute to the actual definition of semantics.

3.2.2 Concrete Syntax

Concrete syntax is concerned with the form of the language [Harel and Rumpe, 2000]. The concrete syntax specifies how abstract concepts are realised in a concrete notation. A language may have multiple concrete syntaxes. Common types of concrete syntaxes are textual, tabular and graphical. The *textual syntax* represents a linear sequence of symbols (e.g. program code). The *tabular syntax* provides a two-dimensional matrix-like view on models (e.g. spreadsheets). Besides x, y dimensions, the *graphical syntax* adds a third dimension z, that enables overlapping ordering of objects. In addition, the other two dimensions are more flexible than tabular (freedom of choosing coordinates, size and colour of objects, etc.). As for the graphical concrete syntax, it may be defined in a static and/or in a dynamic way. The static syntax may be specified using pictorial or vector-based representations. The dynamic graphical syntax may change the visual representation based on the state of the underlying object it represents. A part of the concrete syntax is also a mapping to abstract syntax elements. Each concrete syntax needs to specify how each of the language constructs is represented in that particular notation. As noticed by [Selic, 2007], a mapping must not always represent a direct one-to-one relationship between the concrete and abstract syntax elements. An abstract syntax element may have more than one representational element within a given concrete syntax, or contrary, a single notation element may represent multiple language concepts.

3.2.3 Semantics

The semantics defines unambiguously the meaning of the abstract syntax elements. The semantics is specified by relating abstract syntax elements to concepts of domain whose semantics are already known[6]. Hence, the language semantics consists of a semantic domain and semantic mappings [Harel and Rumpe, 2000], [Karagiannis and Kühn, 2002]. A *semantic domain* describes a specific shared knowledge domain (real-world or computational concepts and phenomena), the language is referring to. The semantic domain may further be divided into static semantics (the what) and dynamic semantics (the how). A *static semantics* describes the meaning of the single concepts and their relationships (semantic schema), i.e. the data that is being processed (by humans or machines). A *dynamic semantics* describes the behavioural part of the language, that is how products of the language (models), behave in "run-time", i.e. the processes handling the data (humans or machines). Dynamic semantics represents a model of computation [Selic, 2007]. According to Selic, a computational model describes

[6]For example, an semi-formal semantic definition of a concept Class may be that it represents a set of objects that share common characteristics. A set is a well-known concept from the set theory of the semantic domain of mathematics.

how the actual computation really occurs. Different computational models exist, including algorithmic models, event-driven models, flow-based models, logic programming models, etc. Note that not all languages have both static and dynamic semantics. Descriptive structure-oriented modelling languages usually do have only static semantics (e.g. ER), whereas process-oriented modelling languages such as petri nets also provide dynamic semantics. A *semantic mapping* defines how elements from the abstract syntax map to the semantic elements. Having the semantic mapping as a first-class language element makes it possible that semantic domains may exist independently of the abstract syntaxes. Hence, semantic domains may be shared by different languages. How semantics are defined depends on the target audience, which may be either human-oriented or machine-oriented. Hence, semantics may be informal (natural language descriptions) or formal (based on mathematics or a machine-interpretable executable formalism).

3.2.4 Interfaces

Languages tend to be less monolithic specifications and more a hybrid set of inter-related languages. Considering this compositional aspect of languages, an additional (optional) element of a language is the notion of a language interface. A *language interface* specifies how a language interworks with other languages. According to Kleppe [Kleppe, 2009] a language may have an optional set of required and offered interfaces. A *required interface* specifies external language concepts needed for a language to work. An *offered interface* exposes a subset of its concepts to be referenced by other languages. The suggested aspect is undoubtedly new, however of fundamental relevance for the underlying work, as it confirms the requirement for having explicit constructs meant for metamodel composition.

3.3 Approaches for Language Definition

In the following, an overview of approaches for language definition is provided structured according to language elements: abstract syntax, concrete syntax, semantics.

3.3.1 Abstract Syntax Specification

Historically, abstract syntax has not been treated separately from the concrete syntax. Abstract syntax was merely an abstract structure to store the result of code (textual concrete syntax) parsing, called abstract syntax tree. Although a tree is a kind of graph, this kind of formalism was not used to explicitly specify the abstract syntax. Therefore, the grammar-based approach is categorised under the concrete syntax specification. The explicit separation of the abstract syntax from the concrete syntax specification gained

attention with an emergence of the model-based approaches to software and system development. Nevertheless, graphs as a formalism to define abstract syntax have been used in the area of visual languages. The common to all approaches is that they treat the abstract syntax (metamodel, graph) as a pivotal element in the language specification. Two major approaches to abstract syntax specification may be identified:

- *Graph-based approaches.* Graphs grammars are rigour formalism to define the abstract syntax of a language. In particular, attributed, directed, typed graphs are suitable for it. Such kind of graph consists of a set of *vertices* connected with *edges*, which both may be given a certain *type* and a set of *attributes* of some data type. Graph *path expressions* and *queries* may be used to specify static constraints and well-formedness rules. The use of graph-based metamodelling has been suggested by Ebert [Ebert and Franzke, 1995]. A visual programming language and environment PROGRES has been developed based on graph grammars [Schurr et al., 1995]. A metamodelling and graph grammar based tool Atom^3 [De Lara and Vangheluwe, 2002] follows the graph-based approach for storing abstract syntax models of languages and the usage of graph grammars for graph manipulations [de Lara Jaramillo et al., 2003].

- *Metamodel-based approaches.* Metamodels, as models of modelling languages, have been recognised as a practical, yet rigour, formalism to define the abstract syntax of a language. Metamodels gained a significant importance in the field of conceptual modelling as well as with the rise of object-oriented design and programming languages, CASE and metaCASE tools and model-driven techniques. A *metamodel* consists of *classes* and *relationships* between classes. *Attributes* of some data type may be assigned to both classes and relationships. A language to define metamodels is called a *metamodelling language*, whereas its model a *meta-metamodel*. Unlike graphs, metamodels, in general, do not provide means for explicitly defining complex constraints and well-formedness rules. Usually, a separate declarative (e.g. OCL [OMG, 2012]) or imperative constraint language is used for that purpose. A multitude of standard and proprietary object-oriented metamodelling languages exist OMG MOF [OMG, 2014], EMF Ecore [Steinberg et al., 2008], CoCoA [Grundy and Venable, 1996], MetaEdit+ GOPPRR [Kelly et al., 1996], ADOxx Meta^2-Model [Junginger et al., 2000], GME MetaGME [Lédeczi et al., 2001], JetBrains MPS Entities DSL [JetBrains, 2013], KM3 DSL [Jouault and Bézivin, 2006]. In Section 3.4 an overview of metamodelling languages is provided.

3.3.2 Concrete Syntax Specification

Approaches to concrete syntax specification may be divided mainly into approaches related to textual and graphical syntax.

Textual Syntax Specification

The approaches for *textual syntax specification* have a long history due to its importance for programming languages. Here, *context-free grammars* is the well-known and widely used approach. Usually, a Backus-Naur Form (BNF) or some derivation of it, is used as a meta-language to specify the textual concrete syntax of a language. At the same time, the grammar represents a mapping to the abstract syntax, since it is parsed to an abstract syntax graph. Another, more recent approach to create a textual concrete syntax is to generate it out of the abstract syntax, i.e. metamodel, based on model transformation [Kleppe, 2007]. This approach has the advantage, since it allows having one abstract syntax specification (metamodel) and multiple textual and graphical syntaxes. The model transformation represents the mapping between the concrete and abstract syntax elements. Besides generating default concrete syntaxes based on metamodels, approaches in model-driven engineering such as EMFText [Heidenreich et al., 2013] allow for the definition of arbitrary concrete syntaxes using an EBNF-like grammar, where each concrete syntax element refers to an element from a metamodel.

Graphical Syntax Specification

In the field of graphical notations, there is no common understanding on how to specify graphical syntax, nor there are widely-used standards as it is the case for the abstract syntax or textual concrete syntax. Historically, the following four approaches are known:

- *Grammar-based approaches.* Grammar-based approaches have been studies in the area of visual languages ([Golin and Reiss, 1990], [Helm and Marriott, 1991], [Wittenburg et al., 1991], [Marriott et al., 1998]). A concrete syntax is specified as a grammar containing a sequence of graphical symbols (bounding box, border, text field, etc.). Using attributed grammars, such symbols may be enriched by structural concepts (abstract syntax elements), giving the "semantics" to the symbols. The symbols are connected using spatial relations to determine positioning (contains, overlaps, left to, right to, etc.).

- *Graph-based approaches.* Graph-based approaches, or more precisely, graph grammar-based approaches are formal techniques to specify graphical syntaxes and have been extensively studied in the area of visual programming languages (VPLs) [Bardohl et al., 1999]. In essence,

it mimics the approach to textual concrete syntaxes by parsing the visual representation of a diagram, which is a directed graph, into an abstract syntax graph and by performing analysis on it. Graph-rewriting techniques are used to analyse graphs. In [Rekers, 1995], a layered graph-grammar based approach is described that separates the graphical representation graphs from the language abstract syntax graphs. Pictorial symbols are assigned to edges and nodes. The spatial relations such as *contains*, *connects*, *touch*, *above* are defined in the spatial relations graph. The abstract syntax graph is derived by parsing the spatial relation graph.

- *DSL-based approaches.* DSL approaches provide a kind of domain-specific language (DSL) for specifying the graphical representation of a language in an imperative way. The graphical DSL utilises vector-based and pixel-based libraries for drawing. The advantage of this approach is a high degree of freedom regarding the notation specification. In addition, both parts of graphical notation, the static and dynamic part may be easily described using imperative languages. The graphical syntax code may be written manually or using visual tools and code generation techniques. Within these approaches, the concrete syntax is usually defined on top of the abstract syntax without explicit abstract to concrete syntax mapping as found in metaCASE tools and metamodelling platforms such as ADOxx [Junginger et al., 2000], MetaEdit+ [Kelly et al., 1996], GME [Lédeczi et al., 2001].

- *Metamodel-based approaches.* Metamodel-based approaches are the most recent group of graphical syntax specification techniques that rely on metamodels as formalism for graphical representation. Similar to graph-based approaches, a separate metamodel is used to specify graphical properties of abstract syntax elements. Hence, a separation of structural and representational concerns exists with an explicit mapping in between. For example, the graphical modelling framework (GMF) for Eclipse implements this approach [Gronback, 2009]. A graphical definition model (representing the graphical concrete syntax formalism) provides concepts such as canvas, rectangle, label, polyline, etc. A mapping model specifies the relation between abstract syntax elements (in this case EMF Ecore model) and graphical notation. Using model transformation and code generation techniques, a graphical modelling tool may be generated. In [Kolovos et al., 2010], a modification of the GMF approach is provided, where a high-level language for direct annotation of the metamodel (abstract syntax) is suggested, to automate the creation of abstract to concrete syntax mapping creation.

Finally, visual, graphical notations must be easily understandable not only by machines (formal aspect) but also by users (cognitive aspect). The ignorance of the representational issues of the graphical syntax has been pointed out in the work of Moody [Moody, 2009]. Moody introduces a systematic, scientific basis for constructing the notation of visual, graphical modelling languages.

3.3.3 Semantics Specification

Specification of semantics is an important part of language design. It is about communicating the meaning of the language to language users (people or machines). The practical purpose of it is manifold. Most obviously, learning about the language and how to use it. On the other side, formal approaches to semantics allow for the formal verification of correctness of the language specification and of the underlying models. Moreover, semantics may be used for automatic generation of interpreters, compilers, tools and mechanisms for the language.

Historically, three most important approaches to formal specification of language semantics exist: denotational, operational, axiomatic [Winskel, 1993]. One additional approach seems to be very practical for domain-specific languages (that rely on code generation) is translational approach [Greenfield et al., 2004], [Kleppe, 2009].

- *Denotational semantics.* Denotation is the term borrowed from semiotics and it means assigning the meaning to signs. In denotational semantics (also known as Scott-Strachey semantics, by the authors [Scott and Strachey, 1971]), syntactic symbols/expressions are assigned mathematical objects. Thus, an abstract syntax of the language is translated into mathematical domain, a semantic domain with well-known meaning.

- *Operational semantics.* Firstly introduced by Plotkin [Plotkin, 1981], operational semantics describes the semantics of the language in that it describes how a valid program is interpreted as sequences of computational steps. Each sequence defines a meaning of the syntactic expressions. Using the inductional techniques, the operational semantics of the language can be checked for correctness. Operational semantics is widely used to specify the dynamic part of semantics. It is a formal specification to assist or generate implementations of compilers and/or interpreters.

- *Axiomatic semantics.* Axiomatic semantics specifies the language semantics by stating assertions about language outcome (models, program) and is used to verify its correctness. It consists of a language able to state assertions (e.g. first-order logic, temporal logic, etc.),

and rules of inference for establishing the truth about assertions. Axiomatic approach for defining semantics of programs was firstly introduced by Floyd [Floyd, 1967][7] and applied for flow chart diagrams. Axiomatic semantics is useful for the verification of correctness for algorithms, by asserting the system state before and after the algorithm executions. This has an interesting effect that two different algorithms will be semantically equivalent provided the same initial and final assertions. Obviously, properties such as efficiency in this case are ignored.

- *Translational semantics.* Translational semantics is more practical and a less formal approach that specifies language semantics by translating a language under study into another language with well-defined known semantics. This approach is particularly useful in the field of domain-specific languages supporting code generation [Greenfield et al., 2004]. A typical example may be a DSL which transforms to a GLP such as Java which again translates to the bytecode, whose semantics are defined by the Java virtual machine[8]. In MDE, the translational approach is usually implemented by defining language mappings and using model transformation approaches [Czarnecki and Helsen, 2006] and code generators.

Another approach for semantics definition is *action semantics* proposed by Mosses [Mosses, 1996] that combines denotational and operational approaches in a pragmatic way, by reducing the mathematical complexity without sacrificing its rigour.

Finally, it must be noticed that the distinction between different approaches is not always clear. Usually a combination of approaches is used to define complete language semantics. However, it may be observed that denotational and translational semantics are usually used for describing static semantics. Operational semantics and translational semantics are used for dynamic aspects of language semantics.

[7] Describing the type of proofs one can do with axioms Floyd mentions [Floyd, 1967],p.19:

> *By this means, we may prove properties of the program... of the form: "If the initial values of the program variables satisfy the relation R_1, the final values on completion will satisfy the relation R_2".*

[8] One may question when this chain of language translations stops? What defines the semantics of the Java virtual machine? It may be again implemented in C++ whose semantics is then translated to assembly, etc. Like in syntax definition, this may end up in bootstrapping, i.e. self-definition. However, as Greenfield [Greenfield et al., 2004] noted, the translation usually stops when there is an agreement of a particular language, that its semantics can be understood by a large enough community that can agree. Therefore, in formal approaches, the language of mathematics is used.

3.4 Overview of Metamodelling Languages

Metamodelling languages are used to describe modelling languages. The process of modelling the modelling language based on metamodels is usually called metamodelling. There is a multitude of metamodelling languages provided, either as standard specifications, or as a core formalism of open source and proprietary metamodelling tools[9].

In this section, an overview of metamodelling concepts is provided. After that, a categorisation framework of metamodelling capabilities is introduced. This is followed by a comparative overview of five metamodelling languages, i.e. meta-metamodels.

3.4.1 Metamodelling Concepts

Metamodel is a fundamental concept of metamodelling. A metamodel is model of a modelling language. A metamodel is a result of metamodelling. Hence, metamodelling is a process and technique of abstraction, classification, and generalisation on the problem domain for which a modelling language or kind of grammar should be defined. Metamodelling is the act and science of engineering metamodels [Gonzalez-Perez and Henderson-Sellers, 2008].

Metamodelling may be organised according to different layered architectures. We adopt the three-layer architecture as described in Section 3.1. The level of models (M1) is used for domain modelling. The level of metamodels (M2) is used for the modelling of languages, i.e. metamodelling. The level of meta-metamodels (M3) specifies the basic constructs for metamodelling and is by definition self-descriptive level. A *model* is an abstracted description of the universe of discourse (real-world phenomenon, logical, physical system, etc.) conforming to the particular metamodel. A *metamodel* is a (abstract syntax) model of a modelling language, conforming to the respective meta-metamodel. A *meta-metamodel* is model of a metamodelling language. Following this linguistic instantiation, in general, a model that resides on the level M_n must *conform* to the particular (meta-)model on the level above it M_{n+1}. A model that conforms to its metamodel is said to be a syntactically valid and well-formed model. A model consists of model elements. A *model element* of a model on the level M_n is an *instance* of a model element on the level M_{n+1}, forming an instance-type relation (classification abstraction). Note that a model element may be an instance of another model element on the same model level. This type of instantiation is called ontological instantiation.

[9]Different terms are used to describe software tools that enable metamodel-based modelling tool development, such as metamodelling tool, metamodelling platform, metaCASE tools, language workbench.

3.4.2 Capabilities of Metamodelling Languages

A meta-metamodel is an abstraction of a problem domain of modelling language definitions, in particular abstract syntax definitions. This problem domain is based on object-oriented paradigm for modelling data structures comparable with entity relationship (ER) models. Hence, there is a consensus about the fundamental set of concepts one metamodelling language should have such as *model*, *class*, *relation*, *attribute*[10]. While this set of necessary concepts is required core for any meta-metamodel, additional elements may contribute to the overall expressive power of that particular metamodelling language in different ways. Metamodelling language constructs contribute to different language capabilities. A capability may consist of a small language feature such as *identification*, which allows unique identification of metamodel elements, or may be a complex set of features such as *modularisation* capability. Basically, one may distinguish language capabilities to those contributing to the core modelling expressiveness and to those enriching the language with productivity constructs. One my refer to the first as a *Meta-language effectiveness* and to the other as a *Meta-language efficiency*. A simple example for a capability of language effectiveness is ability to specify attributes for classes. An example for language efficiency would be the inheritance capability which facilitates the reuse of attributes in class hierarchies, thus boosting up the metamodelling process. Meta-language effectiveness has an impact on the end users of the language, since the constructs decide about the core modelling capabilities. On the other side, the meta-language efficiency does not impact the end user of the language, but the language designer. While a language without meta-language efficiency concepts may have the same impact on the end user, the meta-language efficiency decides about how efficiently the language has been engineered. These concepts help in reducing complexity and facilitation of reuse in language engineering, which is one of the major concerns of the underlying work. Figure 3.2 illustrates a possible categorisation of various metamodelling capabilities. Note that this is not finite categorisation. There may exist other core and supporting capabilities not covered by this framework. For example, meta-language capabilities to describe metalevels and type-instance relationships in multi-level (meta-)modelling [Clark et al., 2014] are not addressed here.

Meta-Language Effectiveness (Core) Capabilities

Metamodelling language effectiveness represents the core group of metamodelling capabilities. It entails *basic constructs* for object-oriented, structural class modelling. These basic constructs may be extended by *role constructs*

[10]Not surprisingly, since model as a data structure is a graph, the core concepts match to the fundamental concepts of attributed typed graphs: graph, vertex, edge, attribute.

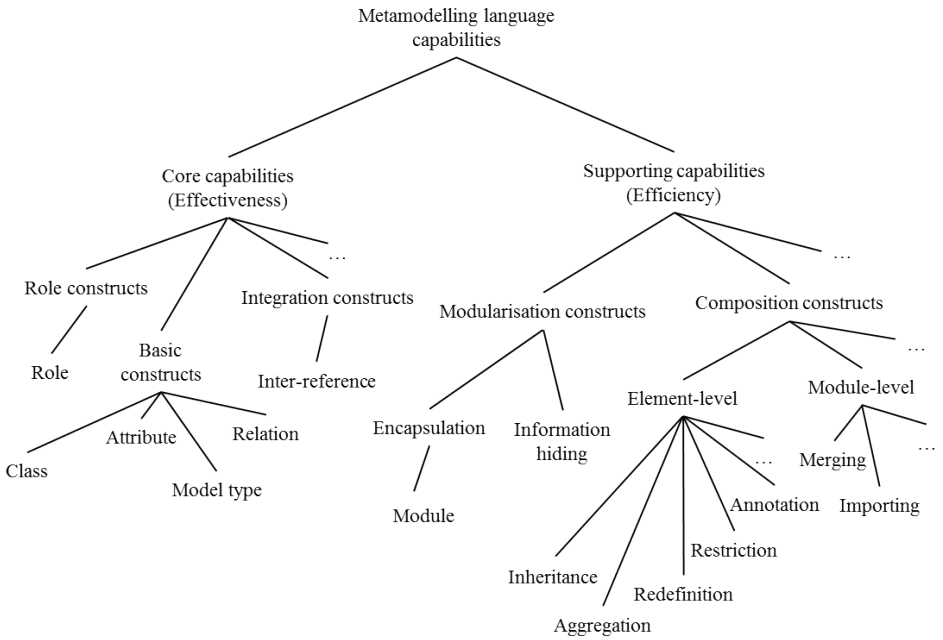


Figure 3.2: Categorisation of metamodelling language capabilities

to support more flexible metamodelling of relations with participating roles. Furthermore *integration constructs* may contribute to hybrid language characteristics, by allowing to create cross-model relations.

- *Basic constructs.* Basic constructs such as class, attribute, relation, model type may be found in all metamodelling languages. A *class* represents a set of objects that share same characteristics such as attributes and relations. It is a type whose instances are metamodel classes on the M2 level, that usually represent entities from a problem domain of a modelling language. An *attribute* describes a property of a metamodel element. An attribute is always of some data type (simple of complex), and has some default value assigned. A simple attribute type is some of the primitive data types such as integer, string, double, boolean, etc. A complex type (also known as a composite type) is a combination of types (simple and complex) that forms a new complex structure. An attribute may describe properties of classes, but also of other basic constructs such as relations and model types. A *relation* describes how classes are related to each other. Relations have arity defining how many classes may participate in a relation, ranging from pairs to n-tuples. Another relation property is multiplicity which describes how many objects may be connected to one object on the opposite side of the relation. If a relation is directed, then connected classes may be grouped to source classes and target classes. A *model*

type describes a set of models with same characteristics. It is a type whose instances are models. A model type groups classes and relations to a logical set of concepts usually describing some aspect of the modelling language.

- *Role constructs.* Role constructs enhance the core construct relation by providing support to more flexibly specify how classes participate in the relation. It introduces the concept of a relation end, also known as endpoint, role, or association end. A *relation end* is an explicit first class element that specifies which classes may be connected to a relation, thus adding an additional level of indirection in specifying relations between classes. A relation may have two or more relation ends depending on its arity. A relation end may also have multiplicities.

- *Integration constructs.* Integration constructs contribute to the integration aspects of modelling languages, to allow inter-connected modelling crossing the model border. Usually, a kind of *inter-reference* concept is used to connect classes directly with model types or classes in different model types or even model types between each other. An *inter-reference* may be realised as a special type of an attribute or as a special kind of a relation concept.

Meta-Language Efficiency (Supporting) Capabilities

Meta-language efficiency subsumes supporting metamodelling capabilities that enhance core capabilities with constructs for more efficient metamodelling. Supporting capabilities contribute to an enhanced reuse of metamodelling artefacts through *modularisation* and *composition.*

- *Modularisation constructs.* Modularisation constructs allow for systematic decomposition of complex metamodels into smaller reusable self-contained *modules* to manage complexity and facilitate reuse. In most of the current approaches, the metaclass model type is (mis)used as a modularisation construct to group classes and relations. However, a dedicated module construct is needed for the encapsulation of arbitrary metamodel elements. In MOF, the concept of package is used for this purpose. Another sub-capability of modularisation is *information hiding.* Information hiding allows a module to hide its internal implementation from its interface specification to other modules. For example, elements within modules may be hidden using access modifiers such as public, private or protected, or using explicit interfaces. Information hiding is currently not supported by none of the metamodelling languages analysed.

- *Composition constructs.* Composition constructs allow for the combination of metamodel elements. Composition constructs provide a kind

of algebra operators for composition. One may differentiate between *element-level* composition operators and *module-level* operators.

- *Element-level composition constructs.* Widely used element-level composition operators that promote element reuse are inheritance and aggregation. In a nutshell, the intention of inheritance is to reuse the structural features of metamodel elements such as properties and references by creating parent-child element hierarchies. A subelement inherits all features of either one superelement (single inheritance) or of more than one superelement (multiple inheritance). *Aggregation* is another form of reuse found in metamodelling approaches. Unlike inheritance, aggregation explicitly establishes a containment relationship between parent and child elements. For example, a class may aggregate a globally defined attribute. As defined in meta-languages such as ADOxx and GOPPRR, aggregation is a Cartesian product function, such that any allowed child element may be aggregated by any allowed parent element. For example, a class or relation may be aggregated and thus reused by multiple model types using a kind of weak parent-child relationship. This feature increases element-level reuse considerably. Further, the *restriction* is another form of composition in which one element restricts the other one. As defined in MOF, restriction can be used only based on inheritance. For example, a relation end might disallow certain metaclass as a connection target of a relation end defined at the super relation end. Similarly, the *redefinition* allows for overriding i.e. redefinition of metamodel element features. This may also be done in combination with metamodel element inheritance. A child element that inherits features from a superelement may also redefine some of its properties. For example, an inherited attribute may have another default value. Finally, the *annotation* is another extensional composition construct which may be used to extend existing meta elements with additional information.
- *Module-level composition constructs.* Module-level composition constructs operate on the level of metamodel modules. An example of a module-level composition construct is a *merging* operator, which merges elements of two modules according to defined rules and constraints. Similarly, an import operator imports elements of one module to the other. Unlike merging, import operator shouldn't cause any structural changes to elements. Importing is comparable to aggregation, but on the level of modules.

3.4.3 ADOxx Meta^2-Model

ADOxx Meta^2-Model is the meta-metamodel of ADOxx, repository-based, multi-language, configurable, metamodelling platform for building domain-specific modelling tools [Kühn, 2010, ADOxx, 2015][11]. ADOxx introduces core metamodelling constructs such as class, relation, attribute and model type. Figure 3.3 illustrates the ADOxx Meta^2-Model. In ADOxx, all meta-model elements are part of a library. A *Library* is a container of all constructs of a metamodel, i.e. of a modelling language. In order to represent different language aspects, library consists of model types (diagram types). A *Model Type* groups classes and relations into aspects and defines a type to which models need to conform to. A model type may have modes. A *Mode* is a model type filtering construct that defines a subset of metamodel elements available in a model type. It is primarily used to reduce the amount of available modelling constructs in a model. A *Class Definition* is an abstract metaclass, that can hold attributes, can be contained by model types, and can participate in relations. A *Class* is a specialisation of the class definition used to define core domain entities in a metamodel. Likewise, a *Relation Class* is a specialisation of the class definition, used to define connections between meta elements. A relation class is a binary relation always having exactly two endpoints. An *Endpoint* specifies which elements may participate in the relation and how (multiplicity). An endpoint allows classes and model types to be target types of a relation and may connect multiple elements, however always of at least one type (either class or model type). Having endpoint metaclass as a metamodelling construct, ADOxx supports the role capability. An *Attribute* represents a property of meta elements, that may have some default value, set of constraints and may be of some specific simple or complex attribute type. All meta elements can have attributes, except attributes themselves. Further, ADOxx supports integration constructs such as *InterRef*, which acts as an inter-reference to either models or objects in other models. InterRef construct is based on the relation class construct. This is an evolution from the earlier versions of ADOxx, in which, InterRef was represented by a kind of pointer attribute. An interref may exist between classes from different model types, between model types, or as a relation between classes and model types. Any combination is allowed. ADOxx supports *reuse by single inheritance* for classes. Furthermore, reuse by aggregation promotes *intra-level reuse* for all meta elements, according to Cartesian product aggregation function. For example, any attribute defined in the metamodel may be reused by any meta element and vice versa.

[11] ADOxx has been built by the BOC Group. An academic version is available for open use as a metamodelling development and configuration platform for implementing modelling methods. ADOxx has been successfully used to develop wide range of commercial modelling tools such as ADONIS [Junginger et al., 2000] which supports standard BPM methods such as BPMN [OMG, 2013], as well as open-source modelling methods and tools in the context of OMILab [Fill and Karagiannis, 2013, OMI, 2015].

Similarly, classes and relation classes may be reused by model types and modes, endpoints by relation classes, etc. That said, any aggregation relation between meta elements represents a *loosely-coupled* containment. Reuse of attributes by aggregation may be considered as a lightweight alternative to multiple inheritance. ADOxx supports basic modularisation in terms of model types and basic composition in terms of inheritance and aggregation. If a single metamodel is represented by a model type, then elements between model types may be reused and extended using inheritance or by aggregating other elements.

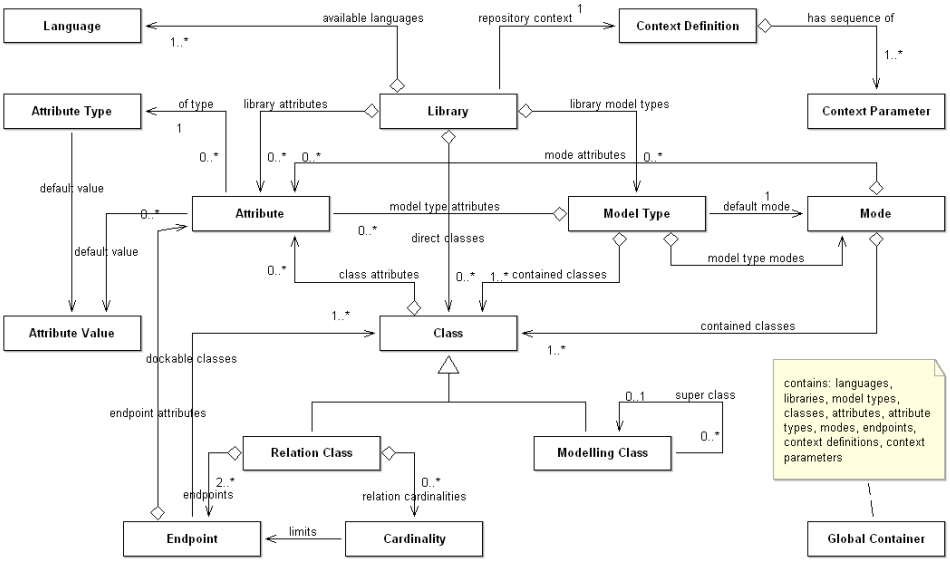


Figure 3.3: Conceptual view of the ADOxx Meta-metamodel: ADOxx-Meta^2-Model

3.4.4 EMF Ecore

EMF Ecore is the meta-metamodel of the Eclipse Modelling Framework (EMF) [Steinberg et al., 2008]. Ecore is a concrete implementation of the OMG EMOF, which represents the essential set of constructs of the MOF meta-metamodel [OMG, 2014] (see subsequent sections). In Ecore, a metamodel consists of a set of modules called *EPackage*. A package consists of classes (*EClass*). A class may have properties which are either references (*EReference*) or attributes (*EAttribute*) of some simple data type (*EDatatype*). Classes may form a class hierarchy using the multiple inheritance relation *eSuperTypes*. References are used to associate classes as binary relations without an explicit role capability (relation end construct). References as well as attributes are owned by classes and cannot exist standalone. Since owned by a class, a reference refers only to a target element type. References may be bi-directional by matching the other reference of

a connected class using the *opposite* property. Also, a reference may be a *containment*, a *container* (opposite of containment) or a *cross-reference*. Thus, a cross-reference contributes to the integration capability. Regarding composition concepts, references and attributes may be reused over inheritance only. The basic modularisation is supported through the EPackage construct. EPackage may contain other classes, and implicitly attributes and references as well as other nested packages. Figure 3.4 illustrates the conceptual view of the Ecore meta-metamodel.

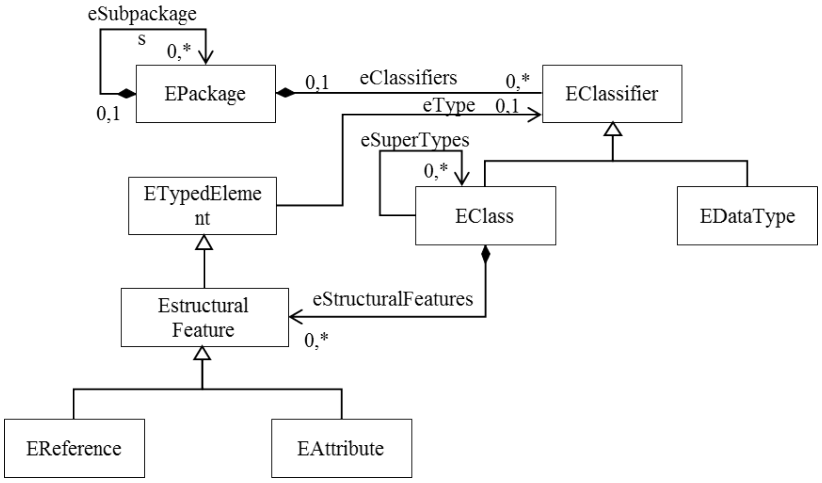


Figure 3.4: Conceptual view of the EMF Ecore meta-metamodel based on [Steinberg et al., 2008]

3.4.5 GME MetaGME

GME MetaGME is the meta-metamodel of the Generic Modelling Environment (GME) [Ledeczi et al., 2001a], a configurable toolkit for creating graphical domain-specific modelling and program synthesis environments developed at the Vanderbilt University, USA. In MetaGME, a metamodel is a paradigm organised as a *project* with *folders*. Nestable folders contain models. A *model* is a compound, container metamodel element containing other metamodel elements over roles. A *role* is an additional level of indirection, holding the information on how a metamodel element participates in a model. In addition, a model consists of aspects. An *aspect* contains a subset of metamodel elements called *parts*. An aspect is a view that contributes to metamodel partitioning based on viewpoints. An *atom* is a primitive metamodel element representing entities (classes) in the problem domain. A *connection* represents a directed or undirected binary relationship between metamodel elements. A connection contains exactly two connection roles/-points (source, target). However, a *connection role* is restricted to specify

exactly one metamodel element as a possible target. Connection role thus supports the role capability, however in a restricted form (binary connection, only one allowed target per role). If more than one element should be connected, this may be done using sets. A *set* owns one or more elements of the same model. Atoms and connections may have *attributes*, which are of some *data type*, which is either simple or an ordered type (enumeration). Connection can be created only between elements that belong to the same parent (model). Cross-model connections are realised using references. A *reference* is a kind of a pointer that can connect to an element in another model or to another model, thus supporting the integration capability. Models, atoms, connections, references and sets represent s.c. first class objects (FCOs). As for modularisation, aspects and roles may be regarded as basic-level encapsulation constructs. MetaGME supports the reuse of attributes using the *multiple inheritance* on atoms, connections and models. The multiple inheritance on the class level allows for the *class equivalence operator*, a union relation, which may connect two source classes and a target class representing the composite of the other two [Ledeczi et al., 2001b], [Karsai et al., 2004]. Unlike other approaches, the inheritance is a connection kind. MetaGME supports two types of inheritance relations, *interface and implementation inheritance*. In addition, the composition capability by aggregation is supported by FCOs through the containment relationship between aspects and parts and through roles. On the inter-language level, MetaGME introduces the concept of a proxy. Basically, a *proxy* references an element (atom, connection) from another metamodel. This *proxy reference* is a kind of weak-aggregation construct. Figure 3.5 illustrates the MetaGME meta-metamodel.

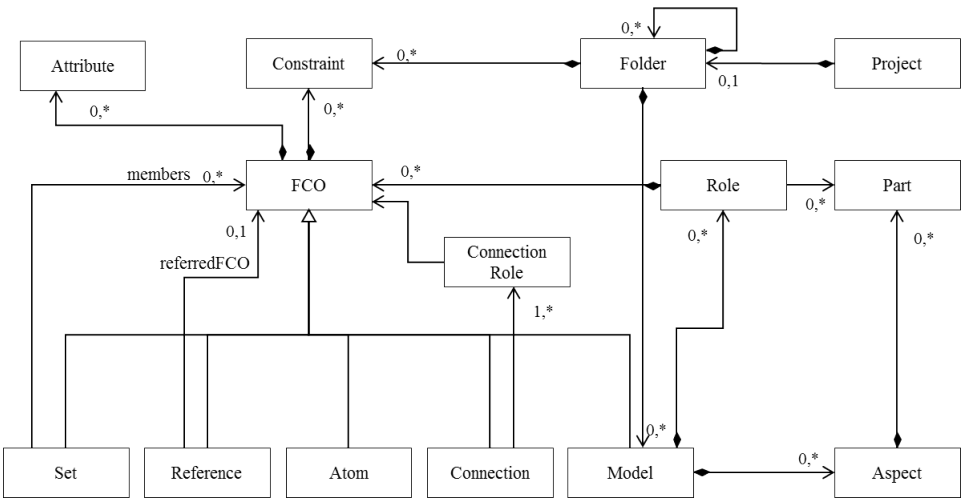


Figure 3.5: Conceptual view of the GME MetaGME meta-metamodel based on [Ledeczi et al., 2001a]

3.4.6 MetaEdit+ GOPPRR

GOPPRR is the meta-metamodel of MetaEdit+. MetaEdit+ is an integrated, repository-based tool set for creating and using modelling languages and code generators developed by MetaCase [Kelly et al., 1996, Tolvanen, 1998, Kelly and Tolvanen, 2008]. The name GOPPRR is an acronym for the first class metatypes of the meta-metamodel, Graph, Object, Property, Port, Relationship and Role. A *graph type* corresponds to a model type whose instances are models. A graph type is a container of other metamodel elements. An *object type* is used to model entities i.e. classes in the problem domain of the modelling language, whose instances are model objects. Object types may be connected via role types and indirectly via relationship types. A *role type* specifies how one or more object types participate in the relationship type via connection. A *relationship type* is related with at least two role types using the *connection* construct. A *port type* defines the possible additional semantics in form of constraints of how the role type may connect to the object type. A relationship type together with composed role types and participating object types forms a *binding*. Hence, a relationship type in GOPPRR is n-ary supporting the role capability. All metamodel elements except bindings may have *properties* of some data type. A property type may be a string, text, number, boolean, collection, or a link to a metamodel element. Collection type is used to define complex data types. Cross-model relationships for integrated modelling are supported in GOPPRR using explosion and decomposition [Tolvanen and Kelly, 2010]. An *explosion* links an object in one graph with another graph type (usually different graph type than the container graph of the source object). A *decomposition* is structurally the same as the explosion. Semantically, an object is linked to a graph type, which has the same type as the container graph of the source object. GOPPRR supports *reuse by single inheritance* for all meta types excluding properties themselves. In addition, *reuse by aggregation* on the intra-level is also supported. For example, a property may be reused by arbitrary meta types, and a meta type may contain many properties. Similarly, multiple bindings may bind same elements. Furthermore, inter-level reuse is supported using the *inclusion*. An inclusion can be defined as an aggregation which can exist only between a graph and its meta elements. Hence, metamodel elements may be reused between multiple graphs. GOPPRR supports only basic modularisation in the sense of graph types and atomic meta elements. The concept of a reusable module with visibility and interfaces is not supported. Figure 3.6 illustrates the GOPPRR meta-metamodel.

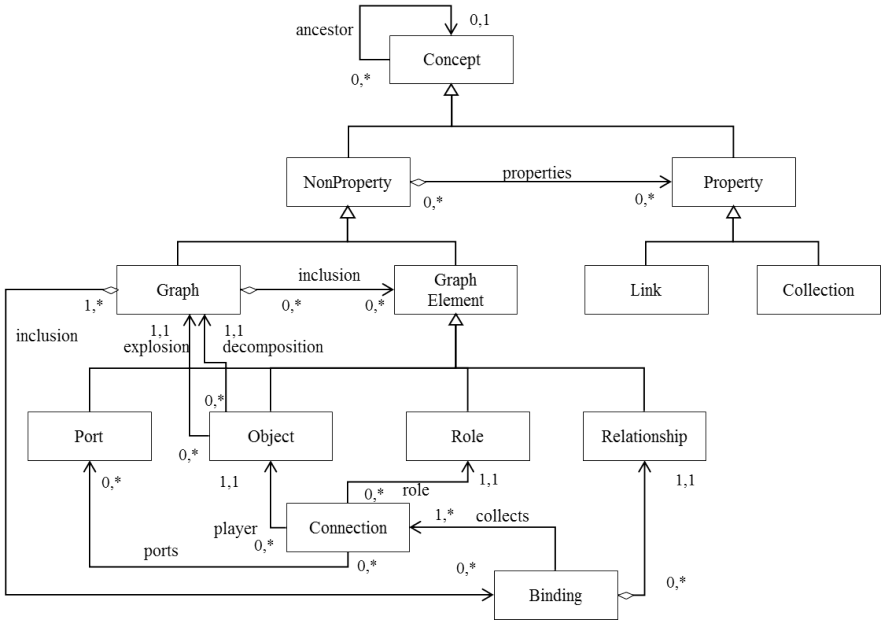


Figure 3.6: Conceptual view of the MetaEdit+ GOPPRR meta-metamodel reconstructed based on [Tolvanen, 1998, Kelly et al., 1996, Kelly and Tolvanen, 2008]

3.4.7 OMG MOF

Meta Object Facility (MOF) is an industry and tool independent standard meta-metamodel defined and managed by the Object Management Group (OMG) [OMG, 2014]. MOF has been used to define UML and other standard modelling languages of the OMG. MOF is divided into the Essential MOF (EMOF) and Complete MOF (CMOF). The EMOF contains only the essential constructs for class-based metamodelling, allowing MOF to describe itself. A concrete EMOF implementation is EMF Ecore. In comparison to EMOF, CMOF contributes with advanced metamodelling concepts for example for reuse and modularisation and definition of metamodels. MOF reuses core constructs of UML2 to define basic metamodelling capabilities for basic class modelling, but also for modularisation via packages. EMOF reuses the *core::basic* package of the UML infrastructure library [OMG, 2011a], including the constructs from the types diagram, classes diagram, data types diagram and package diagram. Description of these elements is comparable to those from Ecore. In addition, EMOF introduces features for metamodelling constructs such as *unique identification* (by URI), *reflection*, *extension*. A reflection adds a link from an element to its metaclass, such that an instance can reflect about itself. An extension tag may be associated to one or more metamodel elements and is used to extend the metamodel with small extensions without heavy-weight changes.

CMOF extends the EMOF with the *core::constructs* package of the UML2 infrastructure library. Regarding the core constructs, CMOF thus resembles semantics for class-based metamodelling from UML2 class diagrams with some constraints. In particular, it redefines the reference to the UML2 navigable *association*, such that a property participates in the association as an *association (member) end*. CMOF constraints the arity of the UML association to be binary. Each association member end property must have at most one type as target and this must be a class. Hence, with classes, properties, associations and packages, CMOF supports core metamodelling capabilities. The role capability is supported via dual semantics of the property as an association end. The reuse of properties (in general structural features) is possible via *multiple inheritance* for classes and associations. Reuse of properties by aggregation is not possible since properties are owned members by its owners. Reuse by aggregation is however possible via packaging. Finally, Figure 3.7 illustrates the core capabilities for class-based metamodelling of the MOF meta-metamodel.

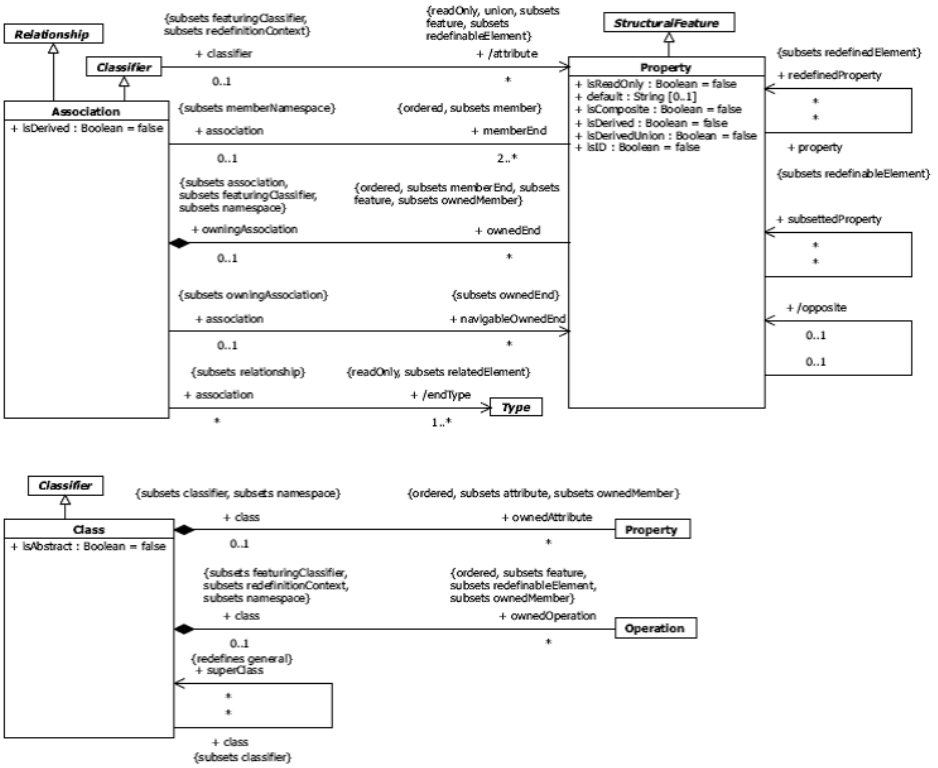


Figure 3.7: Core class-based metamodelling capabilities of the MOF meta-metamodel according to [OMG, 2014, OMG, 2011a]

A *package* represents a modularisation concept that is used to group packageable elements in order to manage complexity and facilitate reuse. A

packageable element that can be an atomic metamodel element or another package is owned by the package using the *packagedElement* association. The *nestedPackage* subsets packagedElement only for packages. A package may import either single elements from other packages as references via *elementImport* or all members of another package via *packageImport*. This feature is derived from the package superclass namespace. Unlike UML2, CMOF does not support visibility of elements within packages. The concept of an interface is not supported. Regarding the composition capability, besides package import, CMOF provides a merge mechanism on the package level that is used to facilitate package reuse. *Package merge*, specified using *packageMerge* association, combines features of a merged package with the merging package to define new integrated language constructs. This way metamodel elements can be extended, instead of using e.g inheritance. Elements in the merging package contain all the features of the same named classes in the merged package. Figure 3.8 illustrates the modularisation capabilities of the MOF meta-metamodel. Further, CMOF includes the redefinition feature of the UML2 to support restriction and redefinition of metamodelling capabilities. Redefinition works in the context of the generalisation. That is, a specialised element may redefine, constrain or subset specific properties (features) of a general element. Redefinition is specified using the *redefinedElement* association between properties. Finally, the aforementioned MOF extension construct *tag* represents a name value pair that can be associated with various elements to annotate them in a lightweight manner. A meta element may be associated with many tags, and the same tag may annotate many elements. A MOF 2.0 in its full version (EMOF and CMOF) has been implemented by the metaCASE tool MOFLON [Weisemöller et al., 2011].

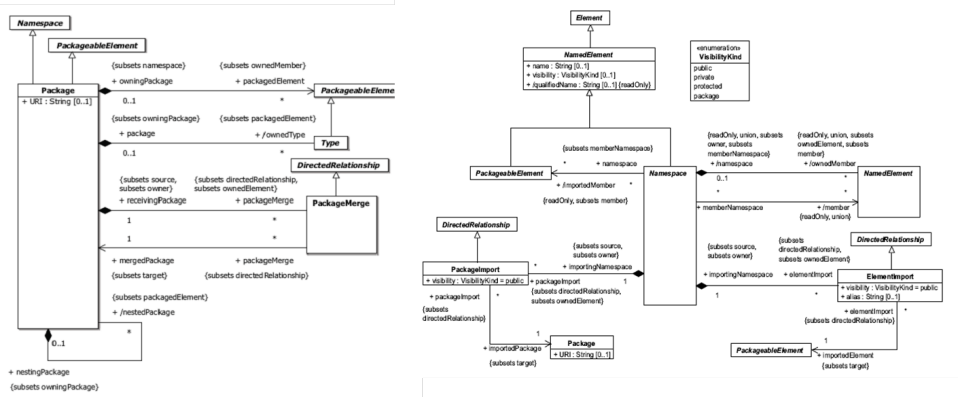


Figure 3.8: Modularisation capabilities of the MOF meta-metamodel according to [OMG, 2014, OMG, 2011a]

3.4.8 GrUML

Graph Unified Modelling Language (GrUML) is a sublanguage of UML and represents a metamodelling language for graph-based modelling [Ebert et al., 1996], [Walter and Ebert, 2011]. GrUML reuses concepts of UML to represent basic graph notions of the TGraph approach [Ebert et al., 1996], that are part of the graph meta-schema such as node, edge, attribute, etc. TGraphs are a general kind of directed graphs whose vertices and edges are typed, attributed and ordered, thus providing sufficient notions to represent metamodels and models. A *GraphSchema* and UML Package correspond to a model type i.e. metamodel, that are used to group metamodel elements. GrUML uses a *VertexClass* as a subclass of a UML class to represent graph nodes, i.e. classes, and an *EdgeClass* as a subclass of UML association to represent graph edges i.e. relations. An EdgeCLass is a binary relationship and it connects exactly two relation ends that are represented by the class *IncidenceClass*. An incidence class thus implements the concept of a *role* by holding the information how a specific class participates in the relation. Incidence class contains information about the aggregation kind, cardinality, direction, etc. VertexClass elements as well as EdgeClass elements may be attributed. An *Attribute* has an attribute value and is of a specific domain/data type. GrUML supports *multiple inheritance* of both classes via *SpecialisesVertexClass* relationship, and relations via *SpecialisesEdgeClass* relationship. Inheritance of edges/relations is a feature that distinguishes GrUML from another formalism such as Ecore where references are represented by class attributes, thus not being able to support neither attributes themselves nor support the relation inheritance. Inheritance of relations is also an additional feature that meta-metamodels such as ADOxx MetaModel do not support. *Reuse by aggregation* is not supported by GrUML. Similar to MOF, a *Package* represents a modularisation concept that is used to group packageable elements in order to manage complexity and facilitate reuse. A package may contain subpackages and *GraphElementClasses*, whose specialisations are NodeClasses and EdgeClasses. Figure 3.9 illustrates the GrUML meta-metamodel.

3.4.9 Comparison of Metamodelling Languages

In the previous sections, various metamodelling language approaches have been analysed according to the metamodelling language capability categories introduced earlier. Although the approaches differ from each other in terminology used, they all are based on object-oriented principles for metamodelling.

- *Core capabilities.* The core capabilities are covered by all approaches. Usually, differences may be observed in the way how relations are modelled considering the ownership, the arity and the role capability.

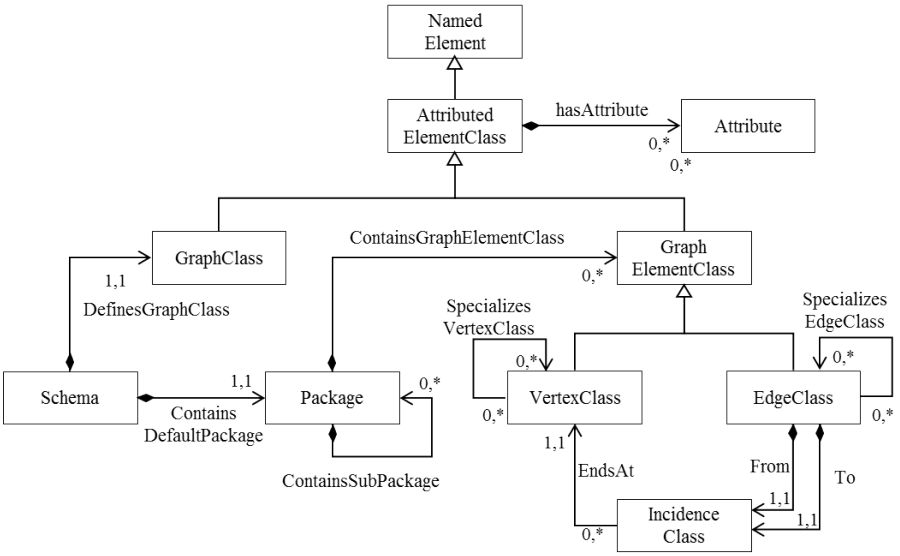


Figure 3.9: Simplified view of the GrUML Meta-metamodel based on [Walter and Ebert, 2011]

- *Integration capabilities.* Integration i.e. cross-referencing capability has been found in all approaches either supported by a special kind of attribute that acts like a pointer to elements in other models, or as a fully-fledged cross-model relation.

- *Modularisation capabilities.* Modularisation capability in terms of basic encapsulation is supported by MOF, MOF-like meta-languages such as Ecore and GrUML. However, information hiding capability is missing. This circumstance may be explained by the fact that the original focus of metamodelling languages was purely on supporting core capabilities.

- *Composition capabilities.* All approaches support the basic composition on the element level via inheritance or aggregation. In addition, MOF as a standard contributes to module-level composition with its package merge and import mechanism. Restriction i.e. property redefinition, and extension tags are also specifics of MOF only.

Finally, Table 3.2 and Table 3.3 provide an overview of the comparison of meta-metamodels with respect to the core and supporting capabilities, respectively.

3.4.10 Other Approaches

There is a multitude of other metamodelling languages, i.e. meta-metamodels emerged over the years in the field of MetaCASE tools, model-driven and

model-based engineering platforms. In the following, acknowledging the fact that this may not be the definite list, some additional metamodelling languages such CoCoA, KM3, MOFLON, Kermeta and Microsoft DSL Tools are briefly summarised.

Complex Covering Aggregation (CoCoA)

CoCoA has been developed in the beginning of nineties as a meta-metamodel for MetaCASE tools for method engineering also known as Integrated Information Systems Engineering Environments (IISEE) [Venable, 1994], [Grundy and Venable, 1996]. The CoCoA metamodelling language is based on the extended ER metamodel. It further extends the ER by the use of collections in roles. A relationship role in CoCoA allowed a role to have more than one possible entity type (endpoint target). CoCoA derives its name from additional ER data modelling concept called Complex Covering Aggregation. Complex covering aggregation represents a composite entity grouping other entities and relationship types, comparable to the notion of model type. With the introduction of entity aliases, entities may be reused between complex covering aggregations thus supporting the language and model integration.

Kernel Metametamodel (KM3)

KM3 is a lightweight textual DSL for metamodel definition [Jouault and Bézivin, 2006]. It has been developed as a practical implementation alternative to MOF meta-metamodel and has been used extensively by the Atlas Model Management Platform to develop multitude of metamodels. By its structure it is very similar to Ecore meta-metamodel. Various M3 level mappings from KM3 to other meta-metamodels have been defined, allowing KM3 to be a bridging structure between different meta-metamodels such as EMF Ecore or Microsoft DSL tools [Bézivin et al., 2005].

MOFLON

MOFLON, a meta-metamodel and a metamodelling tool, is a concrete implementation of the MOF 2.0 used primarily for metamodel-based tool integration [Amelunxen et al., 2006]. While EMF Ecore implements the essential MOF features of EMOF, MOFLON focuses on the complete MOF specification CMOF. Hence, it provides modularisation and extension features of MOF such as package merge, package and element import, as well as property redefinition.

Kermeta

Kermeta is a meta-metamodel and a workbench for metamodel engineering [Drey et al., 2010]. Kermeta meta-metamodel has textual concrete syn-

tax and is built as a compatible extension to EMOF. One of the important extensions to EMOF is the imperative action language, which supports the specification of behavioural semantics of MOF models. Furthermore, Kermeta is aspect-oriented, allowing for simple metamodel extensions using weaving. The weaving technique is mostly used to weave in the behavioural semantics to structural metamodels. Basically, a class can be declared as an *aspect* that may contribute features (attributes, references, properties, operations, constraints) to other classes. The weaving is based on name matching which is comparable to the package merge in CMOF. By weaving aspects one can redefine class features.

Microsoft DSL Tools Meta-Metamodel

As a part of the Visual Studio 2012 IDE [MSDN, 2012], Microsoft provides a set of tools, known as DSL Tools, for the definition of domain-specific languages [Cook et al., 2007]. This tool suite is a realisation of a larger vision of software factories [Greenfield et al., 2004]. The meta-metamodel of DSL Tools supports core metamodelling capabilities. In the MS DSL tools dialect, a *domain model*, a metamodel, consists of *domain classes*, that may have *domain properties*. A *domain relationship* is a specialisation of the domain class, which entails exactly two roles. A *role* specifies how a domain class participates in the relationship. A role may as target also have a domain relationship. A domain relationship may be either an embedded relationship (containment) or a reference relationship (association). Classes and relationships support single inheritance. Regarding the modularisation, it is possible to define a set of abstract domain elements and package them into a s.c. *DSL library*. Other DSLs can then import DSL libraries. No explicit composition operators have been found.

Table 3.2: Comparison of meta-metamodels according to core metamodelling language capabilities

Capability	**ADOxx Meta^2-Model**	**EMF Ecore**	**GME MetaGME**	**MetaEdit+ GOPPRR**	**OMG MOF**	**GrUML**
Core capabilities						
Core constructs						
Class	Class	EClass	Atom	Object type	Class	VertexClass
Attribute	Attribute	EProperty	Attribute	Property	Property	Attribute
Relation	Relation class	EReference	Connection	Relationship	Association	EdgeClass
Relation arity	binary	binary	binary	n-ary	binary	binary
Model type	Model type, Mode	EPackage	Model, Aspect, Role	Graph type	Package	Package
Role constructs						
Endpoint	Endpoint	-	Connection role	Role, Port	Property	IncidenceClass
Endpoint target	multiple	-	single	multiple	single	single
Integration constructs						
Inter-reference	Interref	EReference	Reference	Explosion, Decomposition	Reference	-

Table 3.3: Comparison of meta-metamodels according to supporting metamodelling language capabilities

Capability	**ADOxx Meta^2-Model**	**EMF Ecore**	**GME MetaGME**	**MetaEdit+ GOPPRR**	**OMG MOF**	**GrUML**
Supporting capabilities						
Modularisation constructs						
Encapsulation/Module	-	EPackage	-	-	Package	Package
Info. hiding	-	-	-	-	-	-
Composition constructs						
Element-level	Inheritance (single; class) Aggregation (all metaclasses)	Inheritance (single, multiple; class)	Inheritance (single, multiple; atom, connection, model), Aggregation (atom)	Inheritance (single; all metaclasses), Aggregation (all metaclasses)	Inheritance (single, multiple; class, association), Redefinition (property), Annotation (tag)	Inheritance (single, multiple; VertexClass, EdgeClass)
Module-level	-	-	-	-	Merging (packageMerge), Importing (elementImport, packageImport)	-

3.5 Chapter Summary

In this chapter, we elaborated on the existing work on modelling language engineering in general, and on metamodelling languages in particular. We introduced the basic elements of a modelling language in Section 3.2. In doing so, besides the core elements such as syntax, semantics and notation, we considered the language interface as an element increasingly important for defining hybrid languages. Further, in Section 3.3 we elaborated on approaches for language definition, for each of the core language elements. We concluded that the earlier approaches to language definition were mainly concrete syntax-driven and have not distinguished the abstract syntax from the concrete syntax. The contemporary approaches on the other side target the abstract syntax separately from the concrete syntax and are mainly abstract syntax-driven. We concluded that out of latter approaches, metamodel-based language definition approaches are the most practical. While focusing on metamodel-based language definition approaches, in Section 3.4, we provided an overview of existing metamodelling languages. We contributed with a categorisation framework of metamodelling language capabilities (see Section 3.4.2) and, based on it, provided a comparative analysis of a selected set of metamodelling languages (see Section 3.4.9).

Chapter 4

Metamodelling Environments

"A fool with a tool is still a fool."

Grady Booch

Similar to other engineering domains, engineering of modelling methods and languages requires a dedicated tool set. Although a language may be developed from scratch by programming (hardcoding) it in a software program (e.g. in a dedicated CASE tool), specialised tools exist for language engineering and language use in general. Such environments are known under different names such as metaCASE tools [Alderson, 1991], Computer-Aided Method Engineering (CAME) tools [Kelly et al., 1996], metamodelling platforms [Karagiannis and Kühn, 2002], software factories [Greenfield et al., 2004], language workbenches [Fowler, 2005], modelling frameworks [Steinberg et al., 2008].

In this chapter, we focus on such tools, we commonly call *metamodelling environments*. Metamodelling environments are tools for metamodel-based modelling language engineering and modelling tool derivation. Section 4.1 discusses basic notions of development environments in general. By identifying the commonality of programming, modelling and metamodelling environments, a generic architecture of development environments is provided. Furthermore, a classification of development environments is given based on the their adaptability level. In Section 4.2 we elaborate on metamodelling environments. After the generic architecture of such environments is introduced, we discuss their basic capabilities. The architecture and the capability classification are then used as a framework to evaluate a selected set of such environments. Section 4.3 makes an excursion and reports on ontology-driven software development environments, metamodelling environments enriched by ontology technology. Section 4.4 summarises the chapter.

4.1 Development Environments

Development environments have been recognised as means to improve the productivity and quality in software development [Isazadeh and Lamb, 1997], [Bruckhaus et al., 1996]. As noted by [Isazadeh and Lamb, 1997], according to [Charette, 1986], software development environments provide combined support for software process, method and task automation. Whereas the process defines the main phases and steps in the development to follow, the method defines a paradigm, a language or a set of languages that are used to create development artefacts. The automation is about tools and mechanisms that operate on methods/languages to support the development process[1]. Software development environments, also known as integrated development environments (IDE), are regarded as indispensable sets of development tools, supporting software engineers in their daily work during the complete life cycle of the software development. IDEs consist of a set of tools which provide means for writing, compiling, analysing, refactoring, optimizing, debugging, executing, maintaining, testing and deploying programs written in one or more programming languages[2]. Likewise, modelling environments allow for modelling, analysing, simulating, transforming and compiling, interpreting and executing models defined using one or more modelling languages. Behind both programming and modelling tool sets, modern development environments provide *meta environments* for defining and extending the programming languages and modelling languages and corresponding environments themselves.

4.1.1 Programming Environments vs. (Meta-)Modelling Environments

Independently of the paradigm pursued, development environments follow the same basic set of principles. Usually, such environments are built for a specific paradigm such as programming, modelling or metamodelling. Within a single paradigm, typically, one or more languages may be supported in terms of programming languages, modelling languages. In case of metamodelling, usually, only one metamodelling language is supported. Based on the formalism/language used, the basic unit of processing is a program/source code, a model or a metamodel. The units of processing are compiled to some kind of executable that represents the target outcome of the development process. In programming, a programming language compiler generates machine code for a specific virtual machine. For example,

[1] This definition may be compared to the definition by [Karagiannis and Kühn, 2002] of a modelling method which is a triple of language, mechanisms and process (see Section 2.1)

[2] As noted by [Isazadeh and Lamb, 1997] one of the earliest and simplest attempts to automate certain aspects of the software process in a tool environment was the famous UNIX *Make* utility [Feldman, 1979]

Java compiler generates byte code that can be interpreted by the Java Virtual Machine (JVM) [Oracle, 2013]. In modelling, a model is compiled i.e. transformed, for example, to programming code via code generators, or to a formatted text via model-to-text model transformation [Czarnecki and Helsen, 2006]. In metamodelling, a metamodel is compiled to code [Steinberg et al., 2008] or to configuration files, which both serve to instantiate a language-specific modelling tool. Table 4.1 summarises this comparison.

Table 4.1: Programming vs. modelling vs. metamodelling

Paradigm	**Programming**	**Modelling**	**Metamodelling**
Domain	Application Development	Analysis and Design	Modelling Tool Development
Formalism	Programming Language	Modelling Language	Metamodelling Language
Unit of Processing	Program source code	Model	Metamodel
Environment	IDE	Modelling Tool, CASE Tool	Metamodelling Platform, Meta-CASE
Compiler	Lang-specific Program Compiler	Model Transformer	Metamodel Compiler/Interpreter
Target Outcome	Application	Generated Code, Documents, etc.	Modelling Tool

4.1.2 Elements of Development Environments

Due to the common set of principles as discussed before, and regardless of the paradigm applied, it is possible to identify a common set of elements (a generic architecture) of development environments. Figure 4.1 illustrates such a generic architecture which may be instantiated for (meta-)programming, modelling and metamodelling environments.

It consists of the following basic components:

- *Editors.* Editors provide means for editing work products such as language specifications (abstract and concrete syntaxes, etc.), models, code, configurations, etc. Editors may be either text-based, tree-based, graphical, tabular, form-based, etc.

- *Transformers.* Transformers subsume any kind of transformation tools such as compilers, code generators, model and metamodel transformations, in general tools that allow translation of work products from one formalism into another. For example, a model may be transformed to a text or to a model based on another modelling language.

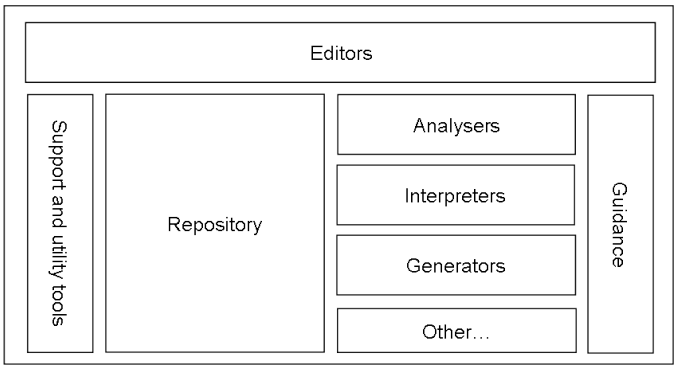


Figure 4.1: Generic architecture of development environments

- *Interpreters.* Interpreters interpret language specifications or models according to defined semantics. Tools like debuggers, simulators, runtime engines and test units are some examples of interpreters. The simulation of business process models is an example of a modelling language specific model interpreter.
- *Analysers.* Analysers help to investigate development artefacts such as code, metamodels, models, etc. Such tools perform different types of static analyses according to a defined set of constraints, such as consistency checking. Metamodel and model querying and tracing, syntax and static semantics validation of models and metamodels may be performed with analysing mechanisms.
- *Repository.* Repository tools store and manage development artefacts such as code, models, metamodels and tool configurations. They abstract from a specific data storage such as a file system or a database. They offer business logic level services to manage development artefacts such as code, models and metamodels.
- *Process Guidance Tools.* Process guidance tools guide engineers according to a defined development process. In collaboration with other tools mentioned before, guidance tools offer help in terms of tasks, check lists, tips and hints on how to proceed in a pursued development process.
- *Process Support Tools.* Process support tools (or utility tools) are vertical tools offering general support services such as collaborative work, versioning, user and security management, etc.

4.1.3 Classification of Development Environments

Development environments in general are built around one special development paradigm, in terms of a programming/modelling language, or a set of

it. Such an environment provides a set of tools to create, edit, analyse and use the artefacts defined by the supported language.

The success of a tool adoption by users depends on how good it serves to solving the problem at hand. The more flexible and adaptable the tool is, considering the problem and context of application, the better its adoption. In particular, the adaptability of the underlying formalism (language) has been the major driver in the evolution of development environments. According to it, three major categories of tool environments may be identified:

- *Language-specific Development Environments (LSDE).* A LSDE supports only one language or a fixed set of languages. In programming, IDEs for C++ or Java programming languages are examples of LSDEs. In modelling, CASE tools have been built for a single specific modelling method or language, notation or a set of notations such as UML, IDEF, SSA, etc. As noted by [Martin, 1994], CASE tools didn't succeed, among others, because of their inflexible capabilities. Users wanted extensions, customisation of their methods, CASE tools could not deliver it due to their hard-wired language support. CASE tools were obviously too rigid, large, complex, and not least, very expensive to produce and customise.

- *Extensible Language-specific Development Environments (ELSDE).* An ELSDE supports a family of similar languages by mapping them to a generic, common language around which the whole toolset is built. Instead of a fixed concrete formalism/language, an ELSDE overcomes the limitation of LSDE by supporting a generic language, which may be specialised by a family of concrete languages. An example of such development environment is Microsoft Visual Studio [Microsoft, 2014b] with its .NET framework [Microsoft, 2014a]. The framework introduces the Common Intermediate Language (CIL) on top of which the basic execution environment, called Common Language Runtime (CLR) is built [Microsoft, 2013]. Arbitrary languages may be mapped to CIL by which interoperability between languages is ensured.

- *Language-independent Development Environments (LIDE).* A LIDE is not bound to any specific formalism, i.e. it supports arbitrary languages. Also known as metamodelling platforms [Karagiannis and Kühn, 2002], software factories [Greenfield et al., 2004], language workbenches [Fowler, 2005], modelling frameworks [Steinberg et al., 2008], LIDEs overcome the deficits of LSDEs by adding a meta-language layer to support the configuration of arbitrary languages. Instead of hard-coding the tools for a specific language, LIDEs provide a tool set for both language design (a meta environment) itself and generic language use. Language definitions are used as specifications to gener-

ate a concrete LSDE either by code generation or by interpretation[3]. MetaCASE tools are LIDEs that are used to generate CASE tools for methodologies/methods based on method and tool specifications. MetaCASE tools solve two most important deficiencies of CASE tools 1) adaptability of language definitions 2) cost of development [Martin, 1994]. Through metamodelling, an arbitrary method/language may be supported. Through automatic tool generation, the cost of development is dramatically reduced. With the emergence of model-driven and model-based approaches to software and system development, and with it, language-oriented development, LIDEs, which provide both language design and use environments, regained a crucial importance for development productivity paired with high flexibility towards changing requirements.

4.2 Overview of Metamodelling Environments

Metamodelling environments belong to the category of LIDEs for modelling language engineering and modelling tool development. The main advantage of such environments compared to the traditional programming or CASE tools is the ability to define an arbitrary modelling language and derive the corresponding modelling tool for a particular modelling problem. Nowadays, a multitude of metamodelling environments for modelling language engineering already exist. Some are focused on textual languages such as domain-specific languages (DSLs) the others on graphical, visual languages for system and business design and analysis. On the other side, some are based on EBNF-like or graph-based paradigms, the others are metamodelling-based. Our focus lies on metamodel-based metamodelling environments such as MetaEdit+ [Kelly et al., 1996], GME [Lédeczi et al., 2001], or the ADOxx metamodelling platform [Kühn, 2004]. In the following, the generic architecture and capabilities of such environments are discussed. Based on thorough capability classification, a selected set of metamodelling environments is evaluated.

4.2.1 Generic Architecture of Metamodelling Environments

Metamodelling environments provide an IDE for both language definition and for modelling. The output of the language definition environment is used as input for the modelling environment, i.e. the modelling environment is derived out of language specification. Both IDEs implement the generic architecture of development environments adapted for metamodelling and

[3]The idea has been initially used to build s.c. compiler-compilers such as YACC [Johnson, 1975]. In a nutshell, a meta-language is used to define the syntax of the target language. The definition of the language is used to configure the generic compiler which results in a generated language-specific compiler

modelling. Hence, such an environment consists of components and tools such as metamodel and model repositories, tools for analysing, transforming and interpreting models and metamodels, various kinds of model and metamodel editors and viewers, as well as a set of guidance and utility tools [Karagiannis and Kühn, 2002, Zivkovic et al., 2009a, Wende et al., 2011]. Figure 4.2 shows a generic architecture of the metamodelling environments based on the corresponding generic architecture of general IDEs. In the following, the single building blocks of this architecture are used as a framework to discuss the capabilities of metamodelling environments.

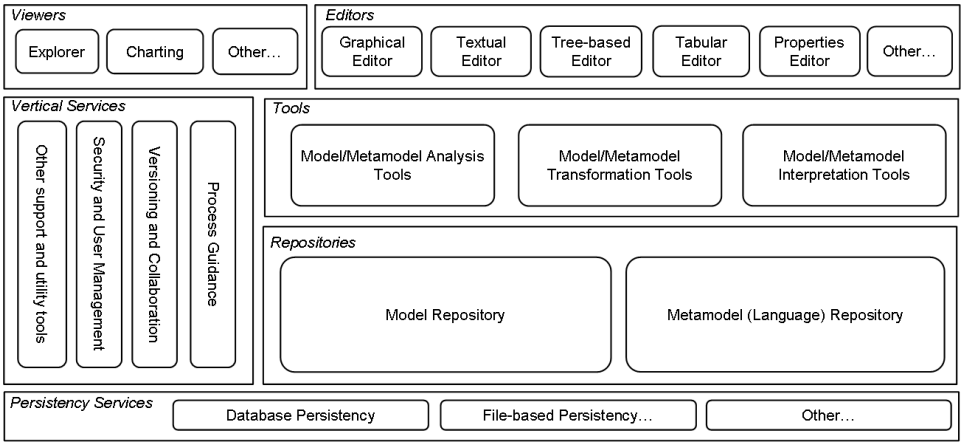


Figure 4.2: Generic architecture of metamodelling environments (capability view)

4.2.2 Capabilities of Metamodelling Environments

Metamodelling environments allow for *deriving modelling tools* out of the language specification. Besides this basic capability, metamodelling environments provide a set of other capabilities which may be categorised according to the metamodelling levels on which they are defined and applied, such as *meta-meta level capabilities*, *metamodel level capabilities*, and *model level capabilities*. In particular, on the metamodel level and model level, meta environment and modelling environment capabilities may be distinguished based on the generic development environment architecture components introduced in the previous section. There are other infrastructure-specific, deployment-specific, implementation-specific components, which are not directly related to the metamodelling capabilities of metamodelling environments, but are rather general to all software systems. For example, a metamodelling environment may be web-based, support cloud-based deployment or just be a desktop application. Although crucial from the implementation point of view, such capabilities will not be considered any further, since this goes beyond the focus of this work.

Figure 4.3 summarises the introduced capabilities of metamodelling environments in a feature tree, as a simplified form of a feature diagram. In the following, capabilities of metamodelling environments are discussed in detail.

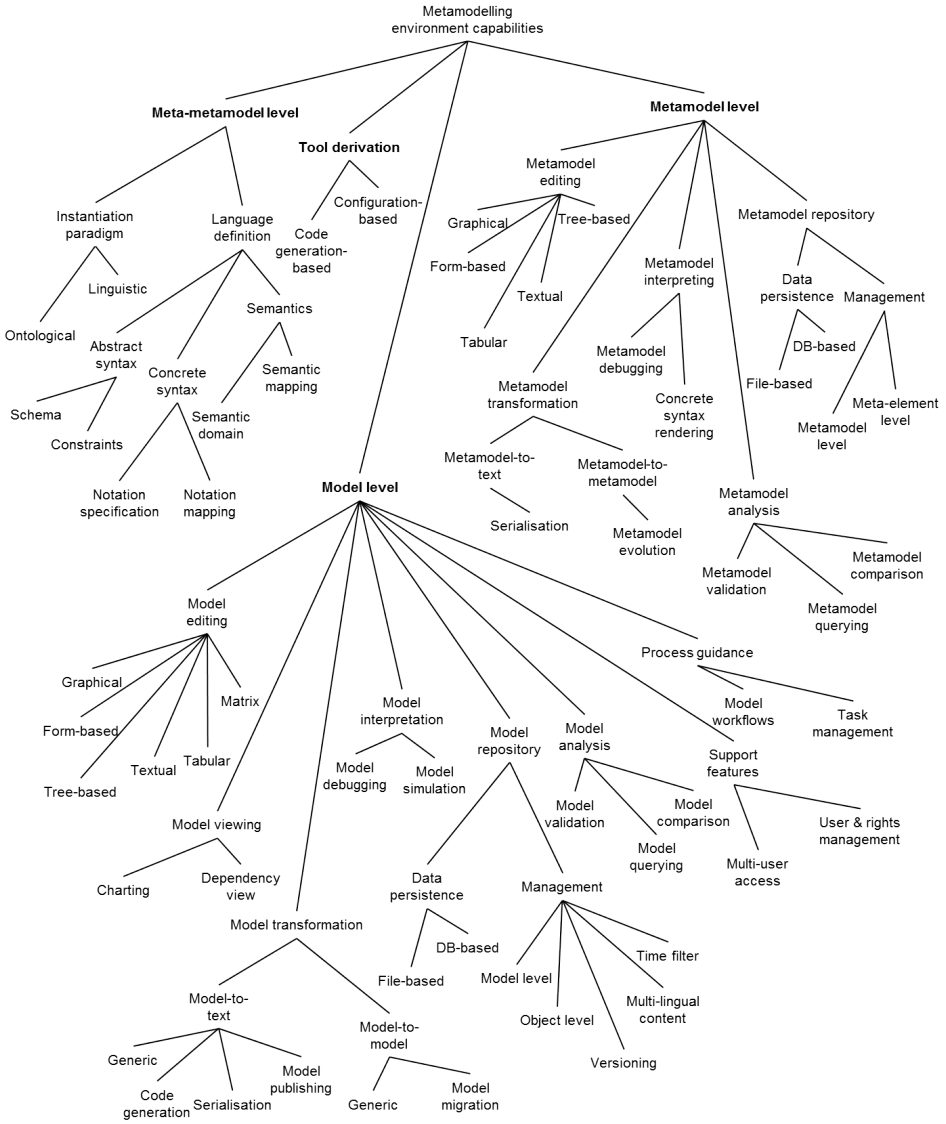


Figure 4.3: Capabilities of metamodelling environments

Modelling Tool Derivation Capability

Derivation of a modelling tool out of language specification is the very basic capability of a metamodelling environment. Modelling tool derivation out of

language specification may be done either by compilation/code generation or interpretation[4].

- *Compiler-based tool generators.* Compiler/Transformation based tool generators generate target code out of language specification usually extending one or more generic underlying frameworks for model-based data storage, graphical or textual editors, etc. Typical example is Eclipse [Steinberg et al., 2008]. The modelling tool must be recompiled upon changes in the language specification.

- *Interpretation-based tool generators.* Interpretation-based tool generators use language specification such as a metamodel as a configuration input for generic tool components. Such environments usually consist of a set of generic components for model-based data storage (repository), generic model editors, query languages which are then "instantiated" for a specific language. Typical examples are ADOxx, GME, MetaEdit+. Usually, the metamodel and model information is stored in the same data source, allowing for a kind of a *live bridge* between the two levels. Due to its "on the fly" execution, changes done on the metamodel level are immediately available on the model level without a recompilation step, thus allowing for more interaction during language development.

Meta-metamodel Level Capabilities

Meta-metamodel level features are related to the basic underlying paradigm and formalism an environment implements in its kernel, i.e. the meta-metamodel. Here, it is sensible to distinguish between *language definition* features and features related to the *language instantiation* architecture.

- *Language definition.* Language definition features reflect single language elements as described in Section 3.2, such as abstract syntax, concrete syntax, semantics and their sub-features.

- *Language instantiation.* Language instantiation is another fundamental property of metamodelling environments as it defines the underlying metamodelling levels and architecture of the tool. Language

[4]The distinction between the compiler-based vs. interpreter-based derivation is not new. In programming languages, program code may either be compiled to a kind of a virtual machine code, or interpreted by a interpreter which then calls the virtual machine-level routines. Traditionally, compiled code promises to be more efficient, since the program is executed directly using the byte-code. Interpreters offer more interaction during the development phase, since the time-consuming compilation step is not needed. However, this difference nowadays diminishes, as both compilers and interpreters integrate features of the other one. For example, compilers may be run in a development mode, in which, the compilation step is greatly reduced and automated without performing long-running code optimisation. Interpreters, on the other side, do a kind of pre-compilation and cashing steps, such that code is not interpreted from scratch every time it is executed.

instantiation differentiates between linguistic and ontological instantiation [Atkinson and Kühne, 2003] as discussed in Section 3.1. Ontological instantiation complements the linguistic one, as the type-instance relation between elements may be supported on each linguistic level. From the implementation point of view, concepts on the meta-metamodel level would need to be added to address this feature. Linguistic type is a common instantiation type for almost all metamodelling environments. Ontological instantiation is rarely supported by today's tools, although case studies justify its need [Zivkovic et al., 2011, Zivkovic et al., 2015].

Metamodel Level Capabilities

Metamodelling environments provide a dedicated tool set for language development. Single features on the metamodel level may be identified by the instantiation of the generic tool architecture on the metalevel.

- *Metamodel editors, viewers.* Language, i.e. metamodel, notation and semantics *editors* and *viewers*, such as graphical, form-based, textual, tabular and tree-based, exist to create and edit language development artefacts.
- *Metamodel transformers. Transformation* features allow for the compilation of metamodelling artefacts into the same or different other formats for the purpose of serialisation, tool derivation, or for a language evolution. A metamodel transformation may thus be categorised in metamodel-to-text and metamodel-to-metamodel transformation approaches.
- *Metamodel interpreters.* Metamodel-based *interpreters* process language definition for the purpose of language debugging or visual notation rendering.
- *Metamodel analysers. Analysis* tools play an important role in language development. Metamodel validation checks metamodels against syntax and semantics rules and guidelines. An ontology-based metamodel validation has been elaborated in [Lekaditis, 2014]. Another metamodel analysis approach based on Prolog has been advised in [Hessellund et al., 2007]. Querying the language definitions using a dedicated query language is especially important when dealing with large projects and multi-metamodel repositories. Metamodel comparison helps finding differences and commonalities in different versions or in completely orthogonal metamodels.
- *Metamodel repository.* Language definition artefacts are stored and managed in a data *repository.* Some tools rely on saving development artefacts in a file system as source code files (e.g. XML-based),

other use databases to persist metamodels and related data. Independently of the persistence strategy, repository management layer offers metamodel management capabilities while abstracting from the specific data layer. Some tools such as ADOxx, MetaEdit+ offer both metamodel and metamodel element repositories for an increased reuse.

- *Metamodelling process guidance.* Metamodelling *process guidance* helps language developers through the language definition process.

- *Metamodelling support tools.* Finally, *process support tools* enable user and rights management, multi-user access and team collaboration features.

Model Level Capabilities

Toolset for model engineering is language-specific and is derived out of the language definition. Similar to the metalevel, single features on the model level may be identified by the instantiation of the generic tool architecture. In fact, in some of the tools these features are implemented by the same components applying the bootstrapping techniques. Hence, on model-level, similar feature set may be identified.

- *Model editors, viewers. Model editors and viewers* play a crucial role for creating, editing and consuming the model information. Ideally, a modelling tool is not restricted to only one type of the editor, but supports various editors such as graphical, textual, tabular, form-based or matrix-based. Viewers may filter and display model information to a problem domain-specific needs. An extensive overview of metamodelling tool capabilities for developing graphical model editors is described in [Dietrich et al., 2013].

- *Model transformers.* Transformation of model information is the cornerstone of model-driven software development [Czarnecki and Helsen, 2006]. Model-to-text transformations are used as code generators or model compilers. Furthermore, they can be used to serialise models to various data formats for the sake of tool interoperability or to formatted, rich texts for publishing and model-based documentation. Model-to-model transformations find usage in model migration scenarios, between different modelling language versions or in model interchange scenarios, between different, but compatible modelling languages.

- *Model interpreters.* Model interpreters execute specific logic on model data based on well-defined algorithms. Most typical example of model interpreters are process-based model simulators. Model debuggers are important for model-based code generation and execution, where each step of the running application can be traced back to the underlying

model. Model interpreters may also be any other mechanisms that compute and derive new data based on model information.

- *Model repository.* Model data is stored in a *repository.* Similar to metamodel-level repositories, data may be persisted in a file-based or db-based storage. Repositories store and manage models, but may also manage model objects in an object repository, which is then available for reuse.

- *Model analysers.* Analysers analyse model information found in model repository for different purposes. Model validation checks models against language-specific modelling rules such as syntax, static and dynamic semantics and modelling guidelines. In [Lemcke et al., 2010,Ren et al., 2013] semantic business process model validation is proposed based on the semantic technology. In [Schwarz et al., 2010] business process models are validated based on expressive traceability technology. Given a dedicated query language, model querying feature may be used internally to implement other transformation mechanisms, interpreters and analysers, and externally by the end users to browse and navigate the models. Similar to metamodel-level, model comparison help finding the differences and commonalities between models. The information is usually displayed graphically, textually, or as a tabular view.

- *Modelling support tools.* Support tools add vertical services to the modelling environment such as user and rights management, multi-user access, team collaboration, etc.

- *Modelling process guidance.* Process guidance aims at guiding the user through the language-specific modelling tasks. There has been an extensive work done on how the process guidance can be realised and integrated into modelling environments based on ontologies and semantic reasoning [Zivkovic et al., 2009b, Aßmann et al., 2013].

4.2.3 ADOxx

ADOxx is an extensible, repository-based metamodelling platform that enables development of modelling tools for a specific modelling method and application domain [Kühn, 2010, ADOxx, 2015][5]. ADOxx is designed according to the principles of product line development, which allows for ef-

[5]ADOxx has been built by the BOC Group. An academic version is available for open use as a metamodelling development and configuration platform for implementing modelling methods. ADOxx has been successfully used to develop wide range of commercial modelling tools such as ADONIS [Junginger et al., 2000] which supports standard BPM methods such as BPMN [OMG, 2013], as well as open-source modelling methods and tools in the context of OMILab [Fill and Karagiannis, 2013, OMI, 2015].

ficient instantiation of new modelling products based on common platform assets. In the following, ADOxx is described considering the metamodelling environment capabilities introduced in the previous section.

Meta-Metamodel Level

ADOxx relies on the three-layer metamodelling architecture, thus the *instantiation paradigm* is linguistic instantiation without explicit ontological instantiation. For the *language definition*, ADOxx implements the meta-metamodel called Meta^2-Model introduced in Section 3.4, which defines the basic abstract syntax constructs and a set of static semantic rules. ADOxx provides built-in extensions of the abstract syntax for the definition of various cardinality constraints, attribute domain value constraints, etc. In addition, complex constraints and validations can be written in a scripting language and triggered upon certain modelling events. The concrete syntax of a modelling language in ADOxx is defined using the concept of a graphical representation (GraphRep). A GraphRep is a graphical symbol that can be assigned to a visible abstract syntax element (visible abstract syntax elements in ADOxx are classes, relation classes and endpoints). GraphRep can be specified textually via the textual GraphRep DSL by utilising the GraphRep drawing library of ADOxx. The GraphRep drawing library contains a comprehensive set of primitives both for drawing symbols and for displaying abstract syntax element properties (attribute values and incoming/outgoing relations). It allows not only for specifying static elements of a symbol (shapes, curves, lines, etc.), but it also enables adding dynamical aspects of concrete syntax e.g. by considering the state changes in object property values (a symbol may change its characteristics based on particular object state such as attribute values). In ADOxx, for one particular abstract syntax definition, one concrete syntax definition may be assigned. The semantics of a modelling language in ADOxx is defined mainly in an imperative way. Besides declarative cardinality constraints, static semantics and operational semantics can be defined using a scripting language or by plugging in external components.

Metamodel Level

ADOxx provides dedicated *editors for metamodelling*. The abstract syntax is edited in a tree-based and form-based editor, while the concrete syntax is edited in textual editor. Concrete syntax (GraphRep) interpreter is provided. *Metamodel interpreter*, i.e. the execution engine for a metamodel is a generic modelling environment itself, which consists of various components such as repository, model editor, analysis and simulation component, publishing, import/export, etc. A file-based *metamodel comparison* is possible. In addition, *metamodel querying* is available via public metamodel APIs,

that can be accessed through scripting. ADOxx supports language evolution via transformation, by providing a possibility to update the productive metamodel with a new version. Language evolution in terms of a metamodel update is possible for all extensive metamodel changes. Changes that may lead to information loss in the repository are not allowed. ADOxx serialises metamodels into a proprietary XML format. Other serialisation formats are available as plug-ins. ADOxx supports db-based *metamodel repository*. Metamodels are packaged into libraries within which metamodel elements are stored in single repositories for intra-metamodel reuse.

Model Level

Modelling tools based on ADOxx are derived based on the configuration and extension of generic modelling components. In the tool derivation process, generic components are configured and/or extended for specific modelling method via metamodel and a set of configuration files. For model creation, editing and viewing, *graphical*, *tabular* and *matrix model editors* exist, as well as *tree-based model and object browsers*. In addition, a set of *charting views* such as gantt, portfolio, radar, dependency analysis view are available to visualise model information graphically. *Transformation of models* in ADOxx is supported and is used for *model serialisation* (e.g. XML), *model migration* and *model publishing*. Model publishing relies on model-to-text transformation which generates formatted text. Since *model interpretation* is always bound to a particular language context (basically operational semantics of the language), ADOxx provides an integrated scripting language with a rich set of metamodel and model APIs to implement language-specific interpreters such as process model simulation. Models in ADOxx are stored and managed in a db-based metamodel-specific model and object repositories. *Model repository* offers, in addition, features such as model and object *time filter*, *model versioning*, *multi-lingual model content support*, etc. ADOxx supports form-based *model querying*, that is executed by a graph-based query engine. It is also possible to write queries based on a proprietary JavaScript-based Query DSL. For *model validation*, structural syntax correctness is ensured by the editors, since they allow only valid constructs (compared with textual editors which basically allow free text input). Additional cardinality and constraint checkers exist, whereas further may be added using extensibility mechanisms. A tabular and graphical *model comparison* may be used to pinpoint differences in models. ADOxx supports a wide range of *support tools*, such as multi-user access, user and rights management, team collaboration, etc. For modelling method specific *process guidance*, ADOxx provides a generic, configurable and extensible model state transition and versioning mechanism, which allows for modelling method specific configuration. Connected to task management and model validation services, it offers a stream-lined process guidance for mod-

elling. Due to its open architecture, ADOxx may be easily extended with additional model-level, generic and modelling method specific features.

4.2.4 GME

The Generic Modeling Environment [Ledeczi et al., 2001a] is a metamodelling tool developed at the Institute for Software Integrated Systems at Vanderbilt University. GME allows for creating domain-specific modelling and program synthesis environments. In GME, a metamodel is a central configuration construct that defines the modelling language of the given application domain.

Meta-Metamodel Level

GME implements the three-layered metamodelling architecture. For the *language definition*, GME introduces its own proprietary metamodelling language MetaGME with an extended UML class diagram notation. The basic constructs for the abstract syntax definition have been described in Section 3.4. The additional syntax *constraints* or static semantics of the modelling language may be defined using the constraint language OCL [OMG, 2012], which is integrated into the platform. OCL constraints are automatically enforced in the target GME instance. Complex and reusable constraints may be defined in constraint functions that may be called from other constraints or other functions, allowing for recursive calls. Function parameters enable the constraint developer to formalise reoccurring definitions in a generic form. The *concrete syntax* is specified via extensions of the UML, mainly in the form of predefined object attributes for constructs such as icon file names, colors or line types. Since there is a one-to-one mapping between the abstract syntax and the concrete syntax, there is no explicit mapping definition between the two. The *instantiation paradigm* is strictly linguistic.

Metamodel Level

GME reuses its generic modelling environment to instantiate the metamodelling environment via bootstraping. Thus, environments on meta and model level share a great deal of features. For language definition, GME offers a *graphical metamodel editor*. However, graphics and symbols for the notation must be created in external tools. Furthermore, a metamodel browser and a constraint definition tool are available for language design. A *metamodel-to-text transformation* takes place to serialise the metamodel to an XML file, which represents a paradigm description, an input for the *derivation* of the paradigm-specific modelling environment. The XMP file has to be then registered in the GME registry before any models of that paradigm can be built. The interpretation of the MetaGME metamodel is performed by the built-in MetaGME *metamodel interpreter*. Furthermore, basic *metamodel*

evolution is supported. It is possible to refine the metamodel and propagate changes on the models. There are built-in model migration operators for a limited number of metamodel changes. *Metamodel analysis* is restricted to the syntax checks and constraint checks. There is no dedicated *query language* for metamodels. *Metamodel comparison* may be realised via the integrated graph-based model transformation engine. GME stores and manages metamodel data primarily in a file-based *repository*. Metamodels may be packaged to libraries and exchanged between metamodel projects to facilitate the reuse and composition. However, *intra-level reuse* of metamodel elements is not given.

Model Level

GME provides a set of generic modelling components that are instantiated for a specific modelling language (paradigm). For model editing, there is an out-of-the-box *graphical model editor* available as well as a *tree-based model browser*. No dedicated model views have been found. For *model transformation*, a GReAT tool suite [Agrawal, 2003] is fully integrated into GME. It allows for various kinds of model transformations. GReAT builds upon the formalism of graph grammars by considering model transformations as graph transformations. Taking metamodels as specification for defining transformation rules, it transforms the input model as the source graph into the output model or text as the target graph. *Model interpreters* of GME translate model data into various forms such as data streams for simulation and analysis tools. GME uses the graph transformation and rewriting techniques of GReAT to implement domain-specific model interpreters [Karsai et al., 2003]. Further interpreters that are domain-independent may be integrated as event-driven add-ons or plugins. Models are stored and managed in a *model repository* either in a file system or in a database. GME does not contain a separate object repository to facilitate reuse of objects between models. However, models can packaged into model libraries and exchanged between different modelling projects, providing that the projects rely on the same paradigm. *Model validation* checks for the correct syntax considering the given OCL model constraints. *Model comparison* may be derived based on the GReAT tool. *Querying of models* is done using a regular expression supported search engine. However, OCL in version 2.0 may be used for model querying, too. GME *support tools* include multi-user access and versioning of modelling projects.

4.2.5 MetaEdit+

MetaEdit+ is a commercial repository-based, integrated metamodelling and modelling workbench for creating and using domain specific modelling languages and code generators [Kelly et al., 1996, Tolvanen, 1998, Kelly and

Tolvanen, 2008]. MetaEdit+ was initially developed as a prototype of a metaCASE tool at the University of Jyväskylä, followed by a commercial version developed by MetaCase [Tolvanen et al., 2007].

Meta-Metamodel Level

MetaEdit+ implements in its kernel the three-layered metamodelling architecture. On the meta-metamodel level, the metamodelling language GOPPRR is provided for the *modelling language definition*. The *abstract syntax* constructs of GOPPRR have been described in Section 3.4. It is possible to define further *constraints* for abstract syntax elements that apply on the modelling level during modelling. There is a limited number of constraint rule forms and templates which can be filled for specific metamodel elements. There is no formal constraint language available for this purpose. The *concrete syntax* can be defined for objects, relationships and roles. Symbols are assigned to respective abstract syntax. Symbols are defined using vector graphics or by importing SVG graphics. There is no explicit language for concrete syntax definition. Also, there is no explicit mapping which assigns one or more concrete syntaxes to an abstract syntax. Definition of a *semantic domain* for abstract syntax elements and the corresponding semantics-to-syntax mapping are not supported. MetaEdit+ supports explicitly the *linguistic language instantiation paradigm*.

Metamodel Level

MetaEdit+ differentiates between the language design and language use environments. Thus, a dedicated metamodelling environment exists featuring a form-based metamodel editor. Form-based metamodel editor consists of a set of so called tools (dialogs) for each of the metamodel concepts. There is an Object Tool, a Property Tool, a Relationship Tool, a Role Tool, a Port Tool and a Graph Tool. For the concrete syntax definition, a symbol editor exists, allowing for both static and dynamic notation definition. In addition to the form-based editor, MetaEdit+ offers a graphical editor offering a proprietary graphical concrete syntax, the GOPPRR notation [MetaCase, 2011], which is similar to the extended ER notation. The graphical metamodel editor is bootstraped out of the modelling environment and integrated on the metamodel level. Graphical editor in metamodelling is used mostly for the early stages of the language development to define the basic structure of the language, due to its limited expressiveness. Form-based editing allow for more complex and detailed metamodelling. Metamodel compilation for tool derivation is needed for graphical metamodels. MetaEdit+ compiles the metamodel to the XML document called MXT which contains the input information for the configuration of the repository-based modelling tool. This XML document is parsed and transformed into modelling

language definition in the repository. If a form-based metamodel editor is used instead, this step is not needed, since both the metamodel editor and the modelling tool are based on the same data source. Thus, MetaEdit+ supports configuration-based tool derivation. Metamodels may be serialised to XML. Metamodel evolution is supported, in a way, that changes in newer metamodel versions do not lead to incompatibilities of models. However, model migration is then a separate required step. Metamodel interpreter is the modelling environment itself with its components. There is a concrete syntax renderer available. MetaEdit+ provides neither an explicit metamodel query language nor a metamodel comparison option. Metamodel data is stored and managed in an object-oriented repository, where single metamodel elements may be reused. MetaEdit+ supports shared multiple user access to metamodel data. Explicit metamodelling process guidance is not given.

Model Level

MetaEdit+ provides a generic modelling environment which is configured by the metamodel and code generator definitions. Models can be accessed and modified by using different model editor types, such as *graphical editors*, *matrix editors*, *tabular editors* and *tree views*. For *model transformation*, MetaEdit+ provides model-to-text transformation for model reporting and code generation. Modelling language-specific *code generators* are configured using the generator definition environment on the metamodel level. There is a scripting language for the definition of transformation rules called MERL, from which the source metamodel elements can be referred. Code generators are run on the model level to transform model to code. A *model interpreter* in MetaEdit+ is realised by the integration with a target program execution environment. By executing the generated code, single code lines may be traced back to models and thus models may be animated/simulated. Other interpreters may be integrated using a rich set of metamodel and model APIs. *Model checking* is provided and is restricted to syntax and constraints checks. Neither an explicit *model querying language* has been found nor a *model comparison* tool. Models and model objects are stored in an *object-oriented db-based repository*. Repository data may be serialised to XML. Also on model level, MetaEdit+ supports *multi-user access* to model repository. Modelling-method specific *process guidance* has not been found.

4.2.6 Eclipse Modelling

Eclipse provides a set of tools to support model-driven engineering. A number of open source projects that contribute to the features of modelling language engineering environments is bundled under the umbrella project called Eclipse Modelling [Eclipse, 2013, Gronback, 2009]. Eclipse Modelling

builds on top of the basic Eclipse programming environment to enrich Eclipse with metamodelling and modelling capabilities. It provides a unified set of modelling frameworks, tooling and standards implementations. Eclipse Modelling as a LIDE follows the compilation-based approach to modelling tool derivation. Modelling language specification is used as an input for diverse generative tools, i.e. code generators which extend underlying generic frameworks with modelling language specific code set.

Meta-Metamodel Level

Eclipse Modelling implements a three-layered metamodelling architecture. The *language definition* is built around the Eclipse Modelling Framework (EMF) [Steinberg et al., 2008]. EMF is a framework for *abstract syntax* development providing an object graph for serialising and deserialising and managing metamodels and models. EMF defines Ecore, the meta-metamodel of the Eclipse Modelling, which has been already described in Section 3.4. OCL [OMG, 2012] language and engine is given to define *abstract syntax constraints*, i.e. static semantics. As a part of the Graphical Modelling Project [Eclipse, 2014d], The Graphical Editor Framework (GEF) and the Graphical Model Framework (GMF) are used to define *graphical concrete syntax*, whereas for the *textual concrete syntax* the Textual Modelling Framework [Eclipse, 2014f] featuring XText is available. Since different frameworks exist for abstract and concrete syntax development, Eclipse Modelling supports more than one concrete syntax to be assigned to the abstract syntax via mappings that are part of concrete syntax frameworks. No corresponding frameworks for *semantic domain* and semantic mappings definition have been found. Eclipse Modelling with its EMF follows the *linguistic instantiation* as an instantiation paradigm. Language evolution is not supported.

Metamodel Level

Eclipse Modelling reuses the modelling environment to provide metamodelling environment via bootstraping. Thus, *tree-based*, *textual* and *graphical* editors exist to define the abstract and concrete syntax of the language. There are no dedicated *metamodel interpreters* defined. On the other side, a rich set of *metamodel transformers* exist, that translate the language specification to other formats needed for *generative tool derivation*. There is no built-in tool for *metamodel evolution*, however an approach for automatic metamodel adaptation and model co-evolution for MOF-conformant metamodels such as EMF has been proposed in [Wachsmuth, 2007]. *Metamodel checking* and *querying* are supported by OCL. *Metamodel comparison* can be performed using the EMF Compare tool [Eclipse, 2014c]. Eclipse-based metamodels are stored by default as EMF XML serialisation in a *file-based*

metamodel repository. There is no concept of a *metamodel element repository* available for reuse.

Model Level

An EMF-based generated modelling environment provides by default a *tree-based model editor*. Depending on the specific concrete syntax framework, *graphical* or *textual* model editors are part of the generated modelling tool. Additional views may be implemented based on GEF. *Model transformation* is the heart and soul of model-driven development, for which extensive support exists. *Model-to-model* and *model-to-text* transformation Eclipse projects offer such tools. For example, the Atlas Tranformation Language (ATL) [Eclipse, 2014b] and the corresponding toolkit may be used for model-to-model transformations, whereas Acceleo [Eclipse, 2014a] may be used for the *code generation*. Eclipse Modelling does not provide out-of-the-box *model interpreters*. However, there is a proposal for a Model Execution Framework [Eclipse, 2014e] based on EMF, that should support development, execution and debugging of models with operational semantics. In the same way as on the meta level, *model checking* and *querying* are supported by OCL, whereas *model comparison* and *model merge* by EMF Compare project. Model data in Eclipse Modelling is persisted by default in a *file-based repository*. However, the CDO [Eclipse, 2014g] toolkit may be integrated as a *db-based model repository* and persistence and distribution framework. With CDO, *support tools* such as multi-user distributed access and collaboration are possible. There is no particular plugin that supports modelling language-specific *process guidance*.

4.2.7 Comparison of Metamodelling Environments

After evaluating the metamodelling environments based on the capability classification framework, it may be noted that all of the environments support the minimum set of necessary features for language definition on meta-metamodel and metamodel level. It may be observed, that on the model level, the amount and focus of the features of the evaluated environments is influenced by the primary environment orientation. While environments such as ADOxx focus on business analysis modelling tools where such tools already represent end-user applications, other evaluated environments set the focus on creation of modelling tools for model-driven development based on code generation where the end-user application is generated out of models and runs outside of the base environment, i.e. in a target run-time environment. A detailed summary on all levels of metamodelling architecture is given in the following. Furthermore, Table 4.2 and Table 4.3 give an overview of the comparison of previously elaborated metamodelling environments considering the feature categorisation framework introduced in

the previous section.

Comparison at the Meta-Metamodel Level

The *instantiation paradigm* that all environments support by default is linguistic instantiation. Interestingly, none of them does support the ontological instantiation, explicitly. As a core feature of an MMLE the necessary constructs for *language definition* such as *abstract syntax* and *concrete syntax* definition are extensively provided. *Constraints definition* seems not to be widely fully supported in a form of a dedicated constraints definition language. As most of the tools focus on one graphical concrete syntax, there is no explicit *mapping specification language* between the abstract syntax and the concrete syntax which would allow for the assignment of multiple concrete syntaxes of the same or different type to one abstract syntax. Explicit support for the *specification of semantics* has not been found in any of the tools.

Comparison at the Metamodel Level

Features on the metamodel level for *metamodel editing* are widely supported. GME and Eclipse reuse the modelling framework to support metamodel editing, too. Contrary, ADOxx and MetaEdit+ provide a dedicated language design environment. Naturally, a *graphical editor* for metamodelling is supported by those tools that bootstrap its environment, whereas the others provide *form-based editors* with a rich set of configuration options. *Tabular editors* for metamodelling are not a common feature. On the other side, only Eclipse Modelling provides an optional *textual editor* for metamodelling. *Metamodel transformation* serves different purposes on the metamodel level. The metamodel-based *tool derivation* based on *code generation* is a basic paradigm of Eclipse Modelling. All other evaluated tools such as ADOxx and MetaEdit+ rely on *configuration-based* tool derivation. *Metamodel evolution*, as an occurrence of the metamodel-to-metamodel transformation is supported by all configuration-based environments. *Serialisation* of metamodels to text, i.e. to a certain XML format for tool *interoperability*, has been found in all environments. Further, *metamodel interpretation* for an increased language design support, apart from concrete syntax rendering, is lacking. However, since the modelling toolset of configuration-based metamodelling environments interprets the metamodel i.e. the language specification, the results of the language design are immediately testable by starting the modelling tool. Code generation tools needs a compilation step in between. Under the category, *metamodel analysis*, *metamodel validation* with respect to the well-formedness of the abstract concrete syntax is provided by all tools. Querying on metamodel level is not extensively supported in a form of a dedicated *metamodel query language* apart from GME

and Eclipse Modelling which use the query part of the OCL for metamodel querying. *Metamodel compare* is either natively supported or it is file-based. The *metamodel repository* in all environments may be *file-based* or *db-based*, although some are focused on db-based data persistence such as ADOxx, GME and MetaEdit+, while providing file-based persistence as an optional feature. In this regard, Eclipse Modelling does the opposite. Driven by the meta-metamodel capabilities, management of metamodel data is supported for all tools on the *metamodel granularity level*, whereas ADOxx and MetaEdit+ go one step further and provide repositories on the *metamodel element* level, as well.

Comparison at the Model Level

On the model level, model editing represents the central feature for all analysed metamodelling environments. While all of the tools provide a variety of *model editors*, ADOxx and MetaEdit+ seem to have the most complete list of model editors. This may be attributed to the fact, that both tools are primarily used for commercial purposes. In addition, ADOxx provides a set of model viewers, not found in other tools. On the other side, Eclipse Modelling is the only environment providing textual syntax and textual model editors with syntax checks, syntax colouring, pretty printing, etc. This can be explained by the application focus of Eclipse in software engineering. *Model transformation* is widely used in all environments. Whereas only Eclipse Modelling provides generic model-to-model and model-to-text transformation, other tools have dedicated languages and engines for specific transformation needs such as *model migration*, *model publishing*, *code generation* and *model serialisation*. Since the *model interpreting* is by nature language and domain-specific, none of the tools provide out-of-the-box model interpreters. Instead, either configurable generic mechanisms may be customised or new ones may be built by using built-in scripting languages and environment APIs. In the category *model analysis*, *model validation* in terms of syntax checks and additional constraint rules is supported by all environments. Further, proprietary or standard *model query languages* such as OCL are also given. *Model comparison* could not be found in MetaEdit+. When comparing *model repository* features, the similar conclusion as on the meta level may be drawn. All tools provide both *file-based* as well as *db-based* data serialisation, where each of the environments supports one of the data sources as primary, and the other as a possible option. Managing models in a repository is a standard feature, for all tools, whereas MetaEdit+ and ADOxx, in addition, provide an *object repository*, allowing for object reuse between models. On the other side, only ADOxx provides features such as the *multi-lingual content* and the *model time filter*. Modelling *support features* are part of all environments. However, in the same way as for db-based model repository, Eclipse Modelling supports it as an optional feature.

User and *rights management* could be found only in commercial tools such as ADOxx and MetaEdit+. *Modelling process guidance* is not supported by any of the modelling environments except in ADOxx. To support this feature, ADOxx features a state-based model workflow engine as well as a basic task management for ad-hoc tasks.

4.3 Excursion: Ontology-driven Software Development Environments

Ontology-driven software development (ODSD) has been an attempt to improve existing model-driven software development (MDSD) by transparently integrating the semantic technology based on ontologies [Pan et al., 2013]. Ontology-driven sofware development environments (ODSDE) extend current MDSD environments (MDSDE) with services based on ontology reasoning. Some of the use cases on applying ontology technology in metamodelling platforms have been discussed in [Živković et al., 2008]. The challenge of building such ontology-enabled MDSDEs is raised by the technological clash between conventional MDSD technology and reasoning technology [Zivkovic et al., 2013]. In the following, it is discussed how ODSDE can address requirements on modern MDSD. After that, a reference architecture for ODSDE is proposed.

4.3.1 ODSD Environments

Ontology-based MDSD aims to adapt scalable ontology technology [Calero et al., 2006] to address current challenges of MDSD [Wende et al., 2009]. Sharing the abstraction level of MDSD metamodelling languages [Happel and Seedorf, 2006], ontologies can be easily integrated and composed with existing metamodelling approaches [Walter and Ebert, 2009]. The integration on the meta-metalevel between the technical spaces is crucial, in order to transform and exchange languages and models between technical spaces. Sharing the common integration infrastructure, the benefits of semantic technology can enhance MDSD environments in several aspects [Aßmann et al., 2013]:

- *Ontology-based software process support.* An ontological conceptualisation of the dependencies, requirements, and results of individual software process steps can be used to formally define MDSD development processes. Having formally defined software process definitions, a reasoning-enhanced software process guidance engine may guide software engineers throughout the development process.
- *Ontology-based development method support.* Not only software processes may be supported via ontology technology, but also development

paradigms. Metamodelling languages enhanced with ontology definitions allow for the definition of metamodel static semantics using logical axioms. Consequently, the semantics of a modelling language is more precise and rigorous, resulting in an increased quality of artefacts specified with such development methods [Walter and Ebert, 2009].

- *Ontology-based repetitive tasks automation.* Services of semantic reasoners such as classification, subsumption checking, and querying may be beneficial in the automation and for the better precision of software artefact validation tools such as model checking, static code analysis and alike.

4.3.2 Reference Architecture for ODSD Environments

The challenge of building ODSD tool environments results from the technological clash between *conventional* MDSD technology and reasoning technology. To address this challenge ontology-enabled tool environments have to provide the bridging technology that can reconcile two different technical spaces. Figure 4.4 proposes a generic architecture for ODSD tool environments that integrates such bridging. This blueprint architecture conforms and extends the generic architecture of metamodelling environments introduced earlier in Section 4.2.1.

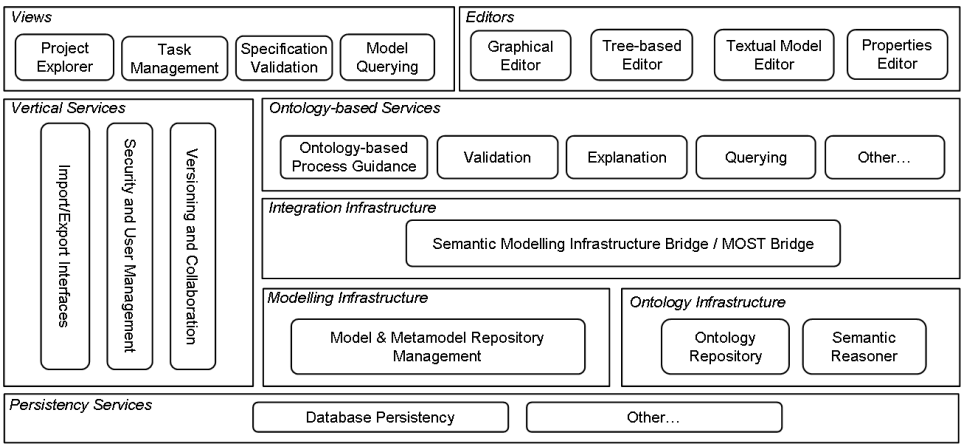


Figure 4.4: The generic architecture for ontology-based MDSD environments

The central building block of the system is the *Bridging Infrastructure* that makes the marriage between the *Modelling Infrastructure* typically found in MDSD tools and the *Ontology Infrastructure* possible. The bridging infrastructure addresses the technical problems of clashing technical spaces, by providing a set of dedicated transformation services. *Ontology-based services* such as *Guidance*, *Validation*, *Explanation*, *Querying* and others are,

therefore, located on top of the *Bridging Infrastructure*. The topmost layer bundles various kinds of *Editors* for creating and editing of model-based system specifications. Besides editors, different *Views* are available that display detailed properties of software development artefacts and results of various analysis tools. Editors and viewers expose services such as guidance, validation, explanation and querying to the developers. The specifics of the ontology technology are transparent to the upper-level components and consequently to the end users. Such abstraction enables seamless application of ontology technology within the technological space of MDSD. In addition, the generic architecture contains *Vertical Services* such as user and rights management or versioning and *Persistency Services*. For further elaboration on single architecture components of ODSD environments, in particular of ontology-based and ontology-enabling services and the bridging infrastructure, the interested reader is referred to [Zivkovic et al., 2009a, Zivkovic et al., 2013, Pan et al., 2013].

Table 4.2: Feature comparison of metamodelling environments - Derivation, Meta-meta and metamodel level

Feature	ADOxx	GME	MetaEdit+	Eclipse Modelling
Modelling Tool derivation				
Code gen.-based	n.a	n.a	n.a	+
Config.-based	+	+	+	n.a
Meta-metamodel level				
Instantiation paradigm				
Ontological	-	-	-	-
Linguistic	+	+	+	+
Language definition				
Abstract syntax				
Schema	+	+	+	+
Constraints	+-	+	+-	+
Concrete syntax				
Notation spec.	+	+	+	+
Notation mapping	-	-	-	+-
Semantics				
Semantic domain	-	-	-	-
Semantic mapping	-	-	-	-
Metamodel level				
Metamodel editing				
Graphical	-	+	+	+
Form-based	+	-	+	+
Tree-based	+	+	+	+
Tabular	-	-	-	-
Textual	-	-	-	+
Metamodel transformation				
Metamodel-to-metamodel				
Evolution	+	+	+	-
Metamodel-to-text				
Serialisation	+	+	+	+
Metamodel interpretation				
Concrete syntax rendering	+	-	+	-
Metamodel debugging -	-	-	-	-
Metamodel analysis				
Validation	+	+	+	+
Querying	+-	+-	-	+
Comparison	+-	+-	+-	+
Metamodel repository				
Data persistence				
File-based	+-	+	+	+
DB-based	+	+	+	+-
Management				
Metamodel level	+	+	+	+
Element level	+	+-	+	-

Table 4.3: Feature comparison of metamodelling environments - Model level

Feature	ADOxx	GME	MetaEdit+	Eclipse Modelling
Model level				
Model editing				
Graphical	+	+	+	+
Form-based	+	+	+	+
Tree-based	+	+	+	+
Tabular	+	-	+	-
Textual	-	-	-	+
Matrix	+	-	+	-
Model viewing				
Charting	+	-	-	-
Dependency view	+	-	-	-
Model transformation				
Model-to-model				
Generic	-	-	-	+
Model migration	+	+	+	-
Model-to-text				
Generic	-	-	-	+
Model publishing	+	+	+	-
Code generation	+	+	+	-
Serialisation	+	+	+	-
Model interpretation				
Model debugging	+-	+-	+	+-
Model simulation	+-	+-	+-	+-
Model analysis				
Validation	+	+	+	+
Querying	+	+	-	+
Comparison	+	+-	-	+
Model repository				
Data persistence				
File-based	-	+	-	+
DB-based	+	+	+	+-
Management				
Model level	+	+	+	+
Object level	+	-	-	-
Versioning	+	-	-	+
Multi-lingual content	+	-	-	-
Time filter	+	-	-	-
Support features				
Multi-user access	+	+	+	+-
User and rights	+	-	+	-
Modelling process guidance				
Model workflow	+	-	-	-
Task management	+	-	-	+

4.4 Chapter Summary

In this chapter, we elaborated on metamodelling environments which represent indispensable means for efficient metamodelling, i.e. modelling language development and the modelling tool derivation. In Section 4.1, we provided a general overview of IDEs, and, in doing so, categorised metamodelling environments as flexible language-independent development environments. Due to their dual nature, such environments provide support for both metamodelling and modelling. While the meta environment is usually bound to a specific metamodelling language, the modelling environment is flexibly derived out of the modelling language definition. These characteristics, the architecture and the capabilities of metamodelling environments have been subject to discussion in Section 4.2. We also provided a comparative analysis of the selected set of state-of-the-art metamodelling environments. Finally, in Section 4.3, as an excursion from the general topic, we discussed how metamodelling environments may be extended by semantic technologies to realise the vision of ontology-driven software development.

Part III

Focus of Work

Chapter 5

On Metamodel Modularisation and Composition

"Having divided to conquer, we must reunite to rule."

M. A. Jackson

The goal of this chapter is to provide an overview of basic concepts for metamodel modularisation and composition in the context of modelling language engineering and to review the existing related work. In Chapter 3, as we introduced the basic notions of modelling language engineering, we agreed that a metamodel represents the pivotal element in the modelling language definition. A metamodel may be developed from scratch as a monolithic artefact or it may be systematically engineered by combining prefabricated metamodel modules that form a new composite metamodel. Modularisation and composition concepts, techniques and tools represent the basis for such modular, compositional metamodel engineering.

We start the chapter by introducing the basic elements of modular, compositional systems (Section 5.1). In Section 5.2, we elaborate on various existing metamodel composition operators. In Section 5.3, we provide a systematic overview of the related work on metamodel modularisation and composition. Section 5.4 summarises the chapter.

5.1 Elements of Modular Systems

According to Aßmann [Aßmann, 2003], a modular, compositional system is a triple consisting of a component model, a composition language and a composition technique. The component model refers to *modularisation*

concepts that are required to describe modules that are to be composed. Under the composition language we subsume *composition concepts* that are needed to specify the composition of modules, i.e. to specify how modules are to be combined to build a composite system. Finally, a *composition technique* deals with mechanisms needed to perform the composition.

5.1.1 Modularisation Concepts

Modularisation as a system engineering technique refers to the possibility to specify specific parts of a system as independent, interchangeable and reusable modules. Modules and modularisation contribute to the separation of concerns by allowing a module to represent only a certain aspect of a system independently from others. Modularisation contributes to reuse and thus to the engineering efficiency, since independent modules may be reused as prefabricated parts for creating new systems. Independency of modules means that, apart from desired and planned, explicit dependencies, a module doesn't interweave with other modules. Modularisation is about *encapsulation* and *information hiding*. In [Booch, 1991], Booch defines encapsulation *"... as the process of compartmentalising the elements of an abstraction ..., the encapsulation serves to separate the contractual interface of an abstraction and its implementation."*. In our terminology, the first part of the Booch definition refers to encapsulation, whereas the second part implies information hiding.

- *Encapsulation.* Encapsulation refers to the compartmentalisation of elements into reusable modules. Usually a kind of a *package* notion exists that explicitly owns its member elements and may refer to elements from other modules. To refer to the elements from other modules, explicit *module dependencies* are required. A module that is not dependent on any other module by design is an *independent*, *self-contained* module. A module that imports elements of other modules by design is called a *dependent* module as it creates an *explicit dependency* on that module. A dependent module creates explicit dependencies to other modules for the purpose of reuse. A module may be reused by arbitrary modules. Reuse of modules may be by copy/value or by reference. *Reuse-by-copy* takes one module and copies it at the place of reuse. The copy of the module is decoupled from its original and it may be changed independently. While this is the biggest advantage of reuse by copy, this independency disallows the exchangeability of modules by new versions, as compatibility may not be guaranteed. *Reuse-by-reference* does not create a copy of the module, but creates a reference to the original module. This form of reuse allows for easier maintenance of modules, since the module has to be exchanged only at one place.

- *Information hiding.* To paraphrase Booch [Booch, 1991], encapsulation serves to decouple the contractual interface of a module from its internal implementation. By introducing an additional indirection level for modules, that of contractual interfaces, a module may hide its internal implementation and expose a subset of its elements only via interfaces. Modules that communicate over stable interfaces are said to be weakly coupled, since their internal implementation is hidden behind interfaces and thus no dependencies exist. There are two types of explicit interfaces, the *required* and *provided* interfaces. Required interfaces are needed for a module to be completed, a provided interface makes the inner elements visible and usable to the outside. A module that relies only on explicit required and provided interfaces is usually called a black-box module. A *black-box* module hides all implementation details of the module and is accessible only via predefined connection points, i.e *explicit interfaces.* An *explicit interface* is defined on purpose during the design of a module. Another level of information hiding is a so called *grey-box.* The term has been introduced by Aßmann to describe program fragments as part of the invasive software composition [Aßmann, 2003]. According to it, grey-box modules define as black-box modules explicit interfaces to the outside, but, in addition, they also introduce implicit interfaces. *Implicit interfaces* are not explicitly defined, but exist based on the known extension points of the underlying language in which the program is written. For example, a java module consisting of one class element, has as implicit interfaces class member sets such as a method set or an attribute set. Thus, without specifying it, during composition, one could add a new method or an attribute to that particular class, without changing any existing semantics of that class[1]. Hence, availability and type of implicit interfaces may vary for different underlying languages. Another form of information hiding may be achieved through *access modifiers.* Access modifiers in programming languages define if and which elements of a module are visible to the outside. This way, a module may control the access to its elements from the outside clients and contribute to the module encapsulation. Visibility properties (aka modifiers) of modules known from general programming languages such as Java or C# may be *private*, *public*, *protected*, *final* or *sealed.* Modules that have explicitly defined visibilities via access modifiers may be called *controlled white-boxes.* Finally, modules that neither restrict access nor expose any kind of interfaces are called *white-boxes.*

[1]However, one might argue, that if some other client code uses metaprogramming techniques to access the meta-information of the program code and rely somehow on the amount of class members of that class, such change might lead to an undefined behaviour on the client side.

5.1.2 Composition Concepts

Composition contributes with concepts to allow for the composition of modules. A composition specification defines how modules are to be combined to build a composite system. *Composition operators* as composition specification constructs are the cornerstone of any composition approach. We can distinguish between *invasive and non-invasive composition approaches*, depending on the way how composition operators affect the modules to be combined.

Composition Operators

Composition can be regarded as a relation between two elements. A *composition operator* is a function that creates a composition relation between elements to be combined. It takes as an input two elements and relates them to denote their compositional structure. The composition relation is directional. Hence, one of the participating elements always plays the role of a *base element* that is combined by the s.c. *composer element* based on particular composition semantics. Composer element is also an element that holds the semantics of the composition. Thus, the base element is said to be *reused* by the composer, which is the purpose of modularisation and composition. In metamodel composition, composition operators operate on the *level of metamodel elements* or on the *level of metamodel modules*. Furthermore, composition operators can be *internal* or *external*. An internal composition operator is part of the underlying formalism (e.g. a programming language or a metamodelling language). On the other side, a composition operator is external if it is part of a dedicated, but external composition language not part of the corresponding formalism (in our case, of a metamodelling language). External composition operators require a separate compiler/interpreter that implements its semantics. For internal/native composition operators, no additional derivation machinery is required. Finally, composition operators may differ in the *coverage* of metaclasses they support. In the case of metamodelling languages, some approaches focus only on the composition of classes, others may allow relations, etc. Ideally, a composition language should support operators for all core metamodelling language elements, as defined in Chapter 3, such as *class*, *attribute*, *relation*, *role* and *model type*.

A detailed overview of existing composition operators is provided in the next section.

Non-invasive vs. Invasive Composition

We defined the composition operation as a function that creates a composition relation from a composer element to a base element. The composer element (module or a single element) is, thus, the holder of the composition

semantics and is adapted to accustom the composition. In derivative techniques, the adaptation of the composer element is done by generating a new adapted copy of the composer element. In interpretative techniques, a new version of the composer element is created to accommodate such change.

The invasiveness in composition determines whether the base element that is being combined needs to be modified in order for the composition to take place. Invasive software composition is the term coined by Aßmann [Aßmann, 2003] to describe a software composition approach that treats reusable program code fragments as grey-box modules and combine them using composition operations such as *adaptation* and *extension*. Invasive software composition adapts and extends components at predefined explicit and implicit hooks by code injection.

Hence, we can distinguish between *non-invasive* and *invasive* composition operations. Figure 5.1 illustrates the difference between non-invasive and invasive composition.

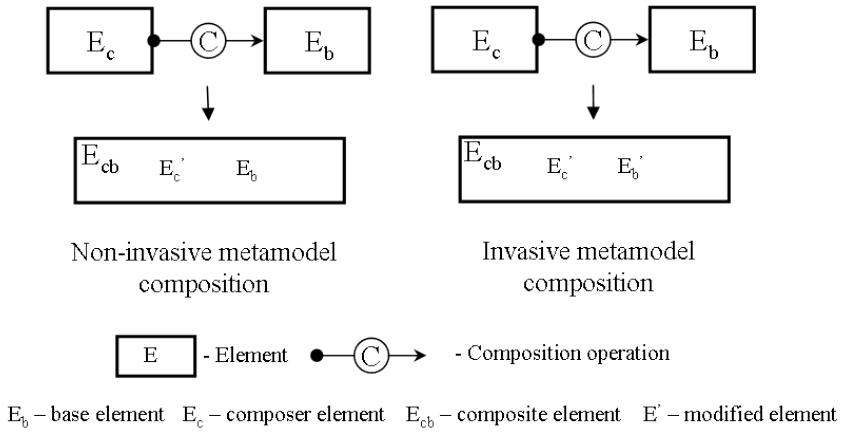


Figure 5.1: Non-invasive vs. invasive composition

- *Non-invasive composition.* In the non-invasive composition, the composer combines i.e. reuses the base element without modifying it. In doing so, the composer element incorporates the semantics of the composition. This is the most common approach for composition. Typical non-invasive composition operators are inheritance, aggregation, importing, etc.

- *Invasive composition.* Invasive composition not only requires changes in the composer element, but it also modifies the base element, the element that is reused, in order to perform the composition. In this kind of composition, the base element to be combined must be a white-box or a grey-box as the composer needs to know the internals of the element (explicit or implicit hooks). We differentiate between *static* and

dynamic invasive composition. *Static invasive composition* imposes alteration of the base element, such that a new version of the base element must be created. It is often applied on the deep copies of elements using the transformational derivation, which then become part of the new composite element. Invasive metamodel merge, as defined in [Zivkovic et al., 2007] creates a new merged metamodel based on existing metamodel parts and may require changes in both metamodels to perform the merge. Alternatively, *dynamic invasive composition* relies on the idea of injecting extensions without explicit modifications of the base element. This technique comes in handy when extending existing metamodel modules and elements. The annotation of metamodel elements as defined in MOF [OMG, 2014] may be considered as a very basic dynamic invasive composition. Tagged values are defined in the composer element. While annotating the base element, the composer element adds tags to the base elements. Semantically, tagged values become part of the annotated elements without explicit base element modification.

It is important to note that both non-invasive and invasive composition operators may be *consistency-preserving* or *consistency-breaking*. For non-invasive operators this applies for the adaptation of the composer element, whereas for the invasive both composer and base elements.

- *Consistency-preserving.* Consistency-preserving composition operators alter the metamodel elements of a module only in an extensional way, thus retaining the semantic compatibility with the original metamodel element. For example, a composer class was extended by a new reference to the element of a base module.

- *Consistency-breaking.* On the other side, consistency-breaking operators apply changes on metamodel elements after which, an underlying instance does not conform to it anymore, i.e. it leads to information loss. An example of such operator may be a class from which an attribute is removed, and instead an outgoing reference is added. These operators deal with compositions of elements of different types (e.g., an attribute should become a reference to class). GeneralizeA2C (attribute-to-class) Operator defined in [Zivkovic et al., 2007] is an example of such invasive operator. In cases where consistency-breaking invasive composition cannot be avoided and a new combined language has to be created or the existing needs to be altered, language *projection* mechanisms are required to allow for seamless language evolution i.e. migration of underlying models. Projection is a mechanism that establishes mappings between the elements of the source metamodel and the target metamodel, based on which, a model transformation may be applied. Projection is out of the scope of the underlying work.

5.1.3 Composition Technique (Derivation of Composite Modules)

Composition technique, as a third element of a modular, composition system, deals with mechanisms needed to derive composite systems out of components and composition specifications. Based on the analysis of the existing approaches, two main derivation directions may be distinguished, *generative derivation* and *interpretative derivation.*

- *Generative derivation* Generative derivation uses model-driven techniques to produce the composite system. It is based on (meta-)model transformation and code generation techniques. This kind of a transformative approach usually subsumes a dedicated, *external composition language* according to which a composition specification, representing the composition glue logic, is created. Taking source modules and the composition specification as input, the generative composition technique generates a new composed module. Each composition directive/operator specifies how it translates to the constructs that conform to the underlying language of a module. For example, an inheritance-like operator will generate a generalisation relationship between a subelement and a superelement in a new module. Some known approaches in metamodel composition that follow this derivation strategy are [Zivkovic et al., 2007], [Weisemöller and Schürr, 2008], [Wende et al., 2010]. A generative composition approach does not create explicit dependencies between source modules, since the composition specification is defined externally, and the composition itself, is performed on the deep copies of source modules.

- *Interpretative derivation* Interpretative derivation interprets composition operators as being part of the underlying formalism (metamodelling language). Unlike generative, transformational approaches, the composition language is natively integrated into the metamodelling language or it may extend it. The metamodelling environment understands composition operators as being part of the language definition itself. Thus, the glue logic is always part either of the extending language or of a new composite language. Due to this fact, neither transformation nor code generation need to take place. For example, the inheritance operator is interpreted with semantics of deriving the properties of a superelement to a derived subelement. For example, the approach realised in the GME tool [Lédeczi et al., 2001] follows the interpretative derivation strategy. Interpretative derivation creates explicit dependencies between modules.

The advantage of the generative approach is that the composition system is decoupled from the underlying formalism. The changes in the composition

language do not need to trigger changes in the metamodelling language, thus, the composition language may be developed independently. However, since it is a two-step process, the results of the composition are visible only after the compilation. Complex compositions may lead to conflicts which sometimes need to be resolved manually as reported in [Wende et al., 2010]. On the other side, the biggest advantage of an interpretative approach is that such inherent composition conflicts may not occur, as the composition directives are interpreted as native language definition constructs. This way, only syntactically valid compositions are allowed, shifting the resolution of conflicting compositions to the design phase.

5.2 Existing Metamodel Composition Operators

In the following, we introduce major types of composition operators as found in the surveyed literature. Besides composition constructs found in mainstream metamodelling languages as introduced in Chapter 3, we also refer to other relevant metamodel composition approaches to broaden the survey. The composition operators are categorised either as non-invasive or as invasive. In addition, we also mention whether the operator is applied on the level of metamodel elements or metamodel modules.

5.2.1 Inheritance

Inheritance is one of the most commonly used element-level composition operators. As known in class-based programming languages and modelling languages, inheritance takes as input two elements and creates a relationship in which one element acts as a *superelement*, and the other one as a derived, *subelement*, inheriting the structural features from the superelement. The superelement is the base element of the composition, the subelement is the composer. Inheritance is a non-invasive operation as only the subelement has to be modified to accommodate the semantics of the composition. Inheritance is used as a basic composition operator in all language composition approaches, due to the fact that it is natively supported by all metamodelling languages such as ADOxx Meta^2-Model, EMF, GOPPRR, GrUML, MetaGME, MOF, etc. Figure 5.2 illustrates the inheritance composition operation.

5.2.2 Redefinition

Redefinition is a special case of the inheritance composition operator. Actually, it is used in the combination with it as defined in MOF. Redefinition extends inheritance in the sense that allows not only to extend the semantics of an element, but also to redefine, constrain or subset specific properties (features) of a superelement [OMG, 2014].

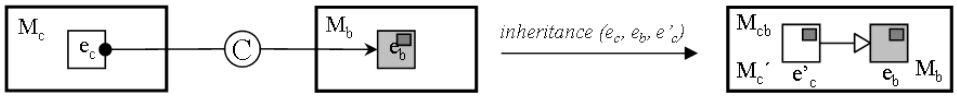


Figure 5.2: Inheritance composition operation: M_c and M_b are source metamodel modules, where e_c, e_b are their respective elements. M_{cb} is a resulting new composite module (a new version of M'_c), in which a new element e'_c, a modified version of e_c (subelement), is created and that inherits features from e_b (superelement) which has been reused.

5.2.3 Aggregation

Aggregation is a *non-invasive* composition operation. It takes as input two elements, *parent* and *child*, and extends the parent element by the child element by means of *part-of* aggregation. The parent element is a composer and the child element is the base element to be combined and reused. The parent element is modified by including the aggregated element, whereas the aggregated element remains unchanged. In ADOxx and GOPPRR, this kind of element composition is used for *intra-language reuse*. For example, an attribute as a first-class element may be aggregated by an arbitrary number of classes. In the same way, a class may be aggregated by multiple model types. Figure 5.3 illustrates the aggregation composition operation.

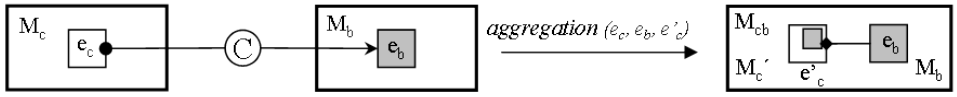


Figure 5.3: Aggregation composition operation: M_c and M_b are source metamodel modules, where e_c, e_b are their respective elements. M_{cb} is a resulting new composite module (new version of composer module M_c), in which a new element e'_c (parent), a modified version of e_c, is created and which aggregates e_b (child) which has been reused.

5.2.4 Merging

Merging takes two elements as a source and combines them together to form a new element as a *duplicate-free union* of the previous two. Merging is a module-level, invasive operator. The merge operator may have *consistency-preserving* character if the changes are of extensional nature. However, merge may be *consistency-breaking*, if any kind of conflicts between element features occurs and needs to be resolved by removing or exchanging elements. The merge operator is used in [Pottinger and Bernstein, 2003] and [Kolovos et al., 2006] to semi-automatically merge models. In MOF [OMG, 2014], the *package merge* is an operator that works on the module/package level and uses name-based matching to merge overlapping elements. The merging (composer) package is extended by the immutable merged (base) package,

leading to a new version of the merging package. Merging is invasive operation. Although the merged package is deep-copied into the merging package thus leaving the source package syntactically unchanged, the operation is invasive semantically, since the merged element now contains extensions from the merging element. In particular, the MOF version of the package merge is used in this way to refine metamodel elements [Selic, 2011]. In [Zivkovic et al., 2007], the Merge rule is applied on metamodel elements to achieve both consistency-preserving and consistency-breaking invasive metamodel integration. Figure 5.4 illustrates the merging composition operator.

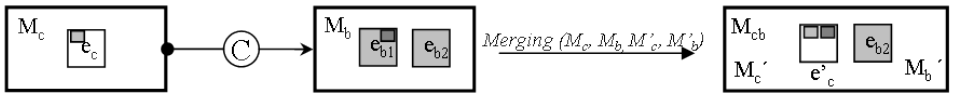


Figure 5.4: Merging composition operation: M_c and M_b are source metamodel modules, where e_c, e_{b1} and e_{b2} are their respective elements. M_{cb} is a new composite module containing the new element e'_c after applying the merge operation on e_c and e_{b1}, and the merged element e_{b2}.

5.2.5 Importing

Importing is a *non-invasive composition operator* that combines contents between modules based on duplicate-free union semantics. The import operator is a module-level operator that takes as an input a set of elements from one module and makes them available in another module *by reference*. The importing module is a composer, the imported module is the base module. In MOF, the *packageImport* and the *elementImport* composition operators are used to import/assign classes and relations of one package into another package. The major difference between the merge operator and the import operator is the reuse mechanism. Merge is based on reuse by copy, import is based on reuse by reference. Furthermore, unlike merge, the import operator does not modify any elements neither of a composer nor of a base module. If the imported elements overlap with existing elements, the import cannot be applied. Figure 5.5 illustrates the importing composition operation.

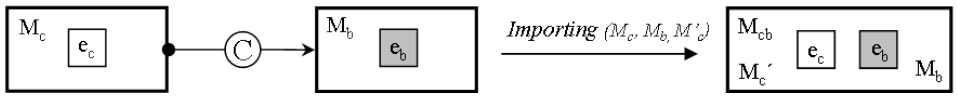


Figure 5.5: Import composition operation: M_c and M_b are source metamodel modules, where e_c, e_b are their respective elements. M_{cb} is the resulting new composite module (new version of composer M_c), which imports the element e_b of M_b using reuse-by-reference. Note, that the composite module contains unchanged elements of source modules.

5.2.6 Template Instantiation

Templating, or *template instantiation* is a *non-invasive composition operator*. It takes as an input a base element which represents the *template* module and a composer which represents the *target* module and *instantiates* elements of a template into a target module based on *reuse by copy*. Basically, template instantiation is a special case of the merge operator as defined in MOF. Instead of importing an existing metamodel part, the metamodel is instantiated according to a given template/pattern [Emerson and Sztipanovits, 2006]. The instantiation process may be done on an existing metamodel element thus forcing the modification of the composer element, or a new element is created. Templating does not create dependencies between metamodel elements to be composed, since it simply generates new elements or adapts existing elements in another metamodel module based on a defined template. In [Weisemöller and Schürr, 2008], templating is used to instantiate parameterisable metamodel components. Template instantiation is done on the module-level, but may also be possible on the element-level. Figure 5.6 illustrates the template instantiation composition operation.

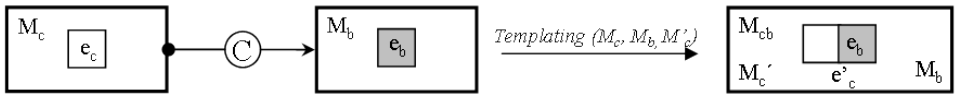


Figure 5.6: Template instantiation composition operation: M_c and M_b are source metamodel modules, where e_c, e_b are their respective elements. M_{cb} is a resulting new composite module (new version of composer module M_c), in which the element e_b (template) has been instantiated to extend the e_c resulting in a e'_c.

5.2.7 Stereotyping

Stereotyping is a non-invasive, element-level composition operator, that allows extensions of base elements. It takes as an input a base element and a composer element which represents the stereotype, and extends the base element by applying the stereotype. In doing so, the composer element, the stereotype, extends the base element with additional structural features, but without inheriting the features of the base element. In earlier versions of UML, stereotyping has been used to apply UML profiles [Fuentes-Fernández and Vallecillo-Moreno, 2004]. Initially, stereotyping was part of the UML language and was referred to as a lightweight metamodelling feature [Selic, 2011] for creating new domain-specific languages based on UML. Since the mechanism was part of the language and not of the meta-language, it was considered as kind of language embedding[2] [Vallecillo, 2010]. However, as of

[2]Language embedding is a special case of language composition, where a new (guest) language is defined using, immutably, the concepts of another (host) language. The

UML 2.4.1 [OMG, 2011a], stereotypes and profiles are part of MOF as well, as they share the same UML infrastructure, which makes the operator applicable for arbitrary languages. In [Langer et al., 2012], the idea of profiles and stereotyping is applied on Ecore, however not on the meta-metamodel level but on the metamodel level through metalevel lifting. The so-called *EMF Profiles* help to customise arbitrary DSMLs that are based on EMF's Ecore.

Figure 5.7 illustrates the stereotyping composition operation.

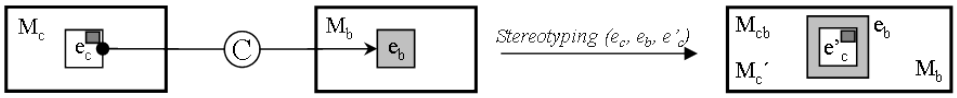


Figure 5.7: Stereotyping composition operation: M_c and M_b are source metamodel modules, where e_c, e_b are their respective elements. M_{cb} is a resulting new composite module (new version of M_c), in which the element e'_c is a composite result of applying a stereotype e_c on e_b.

5.2.8 Annotation

Annotation is an element-level, dynamic, invasive composition operator. A base module is extended by a composer by adding annotation elements to the base module elements. Unlike the inheritance, the idea of annotation is not to introduce a new subelement, but to attach additional information to an existing element without modifying it. In MOF [OMG, 2014], the notion of a Tag is introduced, that represents a name-value pair that can be associated with many metamodel elements. Annotation operation requires modifications on the side of a Tag element, as it holds the information which elements are annotated. Tags have been included to MOF for small, dynamic extensions to reduce the need to redefine the base metamodel. However, since tags are simple string properties, the applicability of this operator is rather limited. Language annotation has been also proposed in [Voelter and Solomatov, 2010] to extend elements by additional properties non-invasively and is implemented in the language workbench MPS [Voelter, 2013]. There, the annotation element is represented by a metaclass *AnnotationNode* which is a built-in construct available in any language module

motivation behind embedding is usually the simplification or a customisation of the more general-purpose host language by introducing one or more derived, more expressive domain-specific languages. That way, one may customise a general-purpose language to the domain terminology, restrict the number of language elements used or add some constraints or syntactic sugar to them while respecting the semantics of the host. Embedding is an old and frequently used technique in the textual DSL engineering, where a small DSL covering specific domain is defined within a general purpose programming language fitting to the syntax of the host [Hudak, 1998]. Embedding facilitates the reuse of the syntax, semantics and tools built on top of a host language and acts as a syntactic sugar of the host language.

that may be extended to define a specific annotation. Figure 5.8 illustrates the annotation composition operation.

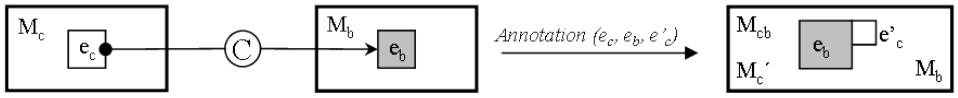


Figure 5.8: Annotation composition operation: M_c and M_b are source metamodel modules, where e_b, e_c are their respective elements. M_{cb} is a resulting new composite module (new version of M_c), in which the element e_b (annotated element) is annotated by e_c (tag element) resulting in a e'_c, since the tag element holds the information about the annotated elements.

5.2.9 Parameterisation

Parameterisation is a transformative composition operator based on subtitutional semantics. It takes two elements, a *place-holder* element (in a composer module) and a *provider* element (of a base module) and substitutes the place-holder by the provider element following the compatibility rules. The place-holder element represents a variable point of the parameterisation function. The provider element is an effective element that parameterises the parameterisation function. Parameterisation is *non-invasive* composition operation since only the place-holder element gets substituted and redefined by the provider/based element. A metamodel module may have predefined subtitutible elements that may be substituted by compatible metamodel elements from another language module. In [Pedro et al., 2008] parameterisation is used to instantiate and combine new domain-specific languages based on existing parameterisable modules using transformative composition techniques. While the provider module may be reused by reference, the place-holder module is reused by copy and adapted to substitute the place-holder elements. In [Weisemöller and Schürr, 2008], a parameterisational approach is used to implement a so called interface binding in transformative way. Here, place-holders are s.c. import interfaces that are substituted by the providers, i.e. export interfaces in a resulting composite module. However, the underlying implementation classes are composed invasively. In [Wende et al., 2010], the role-based approach is introduced based on the role-binding composition operator based on parameterisation semantics. A role represents a place-holder element that may be played by other provider/base elements. Figure 5.9 illustrates the parameterisation composition operation.

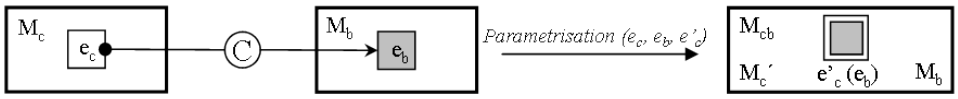


Figure 5.9: Parameterisation composition operation: M_c and M_b are source language modules, where e_c, e_b are their respective elements. M_{cb} is a resulting new composite module (new version of M_c), in which the element e_b (provider element) parameterises e_c (place-holder element) resulting in a e'_c.

5.3 Analysis of Related Work

In the following, an analysis of related work on metamodel modularisation and composition approaches is given. For the evaluation of approaches, we introduce a dedicated metamodel modularisation and composition classification framework.

5.3.1 Classification Framework

The classification framework used for the evaluation of metamodel modularisation and composition approaches should reflect on the basic elements of modular systems as well as to contain all common and variable features found in evaluated approaches. We refer to the elements of modular, compositional systems as introduced in Section 5.1. Therefore, three fundamental elements, the *modularisation*, the *composition* and the *composition technique* represent major feature categories for the evaluation. Each of the main features introduces its own compulsory and optional sub-features a composition system may have.

Figure 5.10 depicts the introduced classification framework as a feature tree, where each feature represents a particular classification category.

5.3.2 Overview of Approaches

Metamodel Modularisation and Composition by Ledeczi et al.

The work on metamodel modularisation and composition found in [Ledeczi et al., 2001b, Karsai et al., 2004, Emerson and Sztipanovits, 2006] describes composition concepts and capabilities of the metamodelling environment GME [Ledeczi et al., 2001a] (see also Chapter 4). The composition technique introduced in GME is interpretative, and the constructs of the composition language are part of the class-based metamodelling language MetaGME (see also Chapter 3). To support modularisation, MetaGME introduces aspects on the intra-language level and projects on the inter-language level. There are no concepts for defining visibility of modules, all modules are white boxes. To allow for composition, MetaGME introduces the concept of a proxy. Basically, a proxy references an element (atom or connection) from

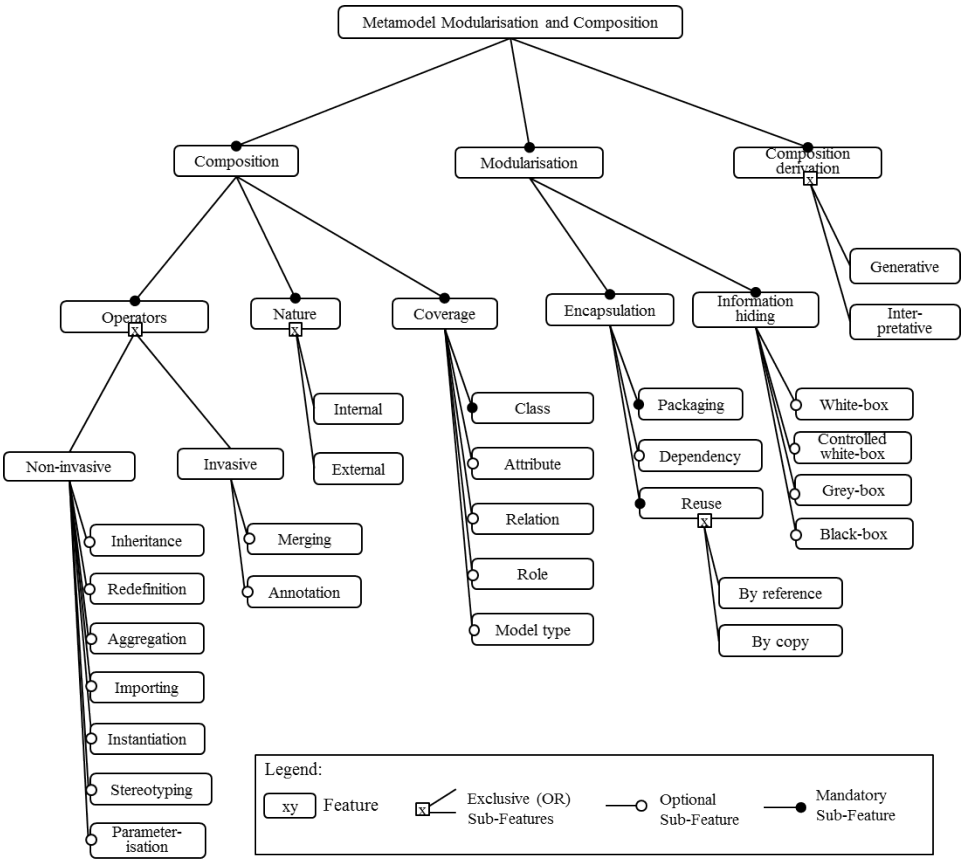


Figure 5.10: Classification framework for the evaluation of metamodel modularisation and composition approaches

another metamodel. The proxy reference introduces explicit dependency between meta-models and may be regarded as a manifestation of the *importing* composition operator. Further, MetaGME reuses existing UML-based constructs such class *inheritance*, but also introduces special types of inheritance relationships such as implementation inheritance and interface inheritance to support non-invasive composition. In addition to standard composition operators, a dedicated class equivalence operator is introduced. The class equivalence operator connects two base classes and a target composer class representing the composite of the other two. This kind of equivalence operator with union semantics is achieved using the *multiple inheritance* between the two source classes as superclasses and the composite class as a subclass which inherits all properties of two source classes. Template instantiation is another composition operator supported [Emerson and Sztipanovits, 2006]. Instead of importing an existing metamodel part, a metamodel is instantiated according to a given template. Other known composition operators such as merging, redefinition, aggregation, stereotyping, annotation and pa-

rameterisation are not supported.

Metamodel Modularisation and Composition in MOF and UML

Meta-metamodel standard MOF [OMG, 2014] introduces a basic set of concepts for metamodel modularisation and composition. To support modularisation, MOF features the notion of a nestable package. However, elements in metamodel packages may only be defined as public, white boxes. Besides inheritance, MOF supports the importing operator via packageImport and elementImport. Furthermore, packageMerge follows the reuse-by-copy to implement the merge operation. Package merge as defined in MOF is very similar to the semantics of generalisation and inheritance. Merging extends the source element with structural properties of the target element such that the source element contains the properties of both. As noted by Selic [Selic, 2011], package merge allows for refinement, because one can define an element with the same name, add some properties and merge them back to the package where the element to be extended resides. The restriction is that it works based on name matching and may be applied only on the package level. The package merge operations in UML2 have been analysed and formalized in Alloy [Zito and Dingel, 2006,Dingel et al., 2008]. Further, MOF supports stereotyping with UML Profiles which are part of the UML Infrastructure specification [OMG, 2011a] that is shared by MOF and UML. Another composition capability supported by MOF is annotation. The annotation operator in MOF is represented by the *Extension*, a lightweight approach to annotate existing metamodel elements with *Tags* that solely represent simple name-value pairs. Aggregation and parameterisation composition operations are not part of MOF.

Metamodel Components in MOF 2.0 by Weisemöller and Schürr

In [Weisemöller and Schürr, 2008] an extension for the MOF 2.0 is proposed, suggesting concepts for the specification of metamodel components beyond packages and for the binding-based metamodel composition based on parameterisation operator. Weisemöller and Schürr propose the notion of a metamodel component which may have internal and external interfaces, thus supporting black-box encapsulation. The notion of visibility is not given. If a metamodel component has import interfaces, that means that within that component there should already be placeholders where a provided interface from other component should be bound to. The binding of provided and required interfaces is done using an additional interface binding metamodel composition operator based on parameterisation. The approach deals only with the composition of class elements on the level of names. The derivation has transformational, generative character.

Metamodel Modularisation and Composition in GOPPRR

The metamodelling capabilities of the metamodelling language GOPPRR [Tolvanen, 1998,Kelly and Tolvanen, 2008] have been discussed in Chapter 3. Here, compositional aspects are summarised. As for the modularisation, the concept of graph types may be regarded as a kind of a basic modularisation concept in GOPPRR. Core reusable elements such as object types and relationship types may be grouped inside of a graph type, in order to define elements of a metamodel. However, the concept of a reusable module/package with visibilities and interfaces for richer encapsulation is not supported. As mentioned, an inclusion can be defined as a kind of importing composition operation which can exist only between a graph type and meta elements. Hence, metamodel elements may be reused between multiple graphs from the global pool of meta elements. Other supported composition operations are inheritance and aggregation. The language derivation is purely interpretative, since composition constructs are part of the metamodelling language.

Metamodel Modularisation and Composition in ADOxx Meta^2-Model

The metamodelling capabilities of the ADOxx metamodelling language have been discussed in Chapter 3. Here, modular and compositional aspects are shortly given. ADOxx provides basic modularisation within a metamodelling project (library). Metamodel elements are grouped into model types defining the aspects of an integrated modelling language. Metamodels are therefore white-boxes. As for the composition, elements may be composed using inheritance and aggregation. Similar to GOPPRR, ADOxx supports reuse of classes, attributes, relations and roles by aggregation, as a powerful but lightweight alternative composition construct to multiple inheritance. The basic language derivation strategy is interpretative.

Metamodel Modularisation and Composition in EMF Ecore

As an implementation of the MOF essential constructs, EMF Ecore supports only a subset of constructs for modularisation and composition (see Chapter 3). The basic modularisation is supported through the package construct. A package may contain other classes, attributes and references as well as other nested packages as white-boxes without information hiding. Concerning the composition, basic composition operations between packages such as inheritance are possible. The basic importing composition operation is however available on the Ecore project level by importing the ecore model (metamodel) as a so called resource. That way, the whole metamodel with its packages can be reused by another metamodel for extensional composition. The language derivation strategy is interpretative.

Metamodel Parameterisation by Pedro et al.

In [Pedro et al., 2008] the idea of parameterisable metamodel fragments is introduced for the purpose of DSML prototyping. In this approach, a metamodel is called parametric if it has "holes"/parameters (required interfaces). Contrary, a metamodel is called effective if it "fills the holes" of the parametric metamodel. The supported composition operation is parameterisation, which is based on the compatible substitution operation that replaces an existing element in the metamodel by another compatible one according to the provided element binding. The approach, however, combines the parameterisation with other methods such as association and merge. Ecore is used as the underlying metamodelling formalism. Hence, the basic modularisation concept is a white-box Ecore package. The derivation technique is transformational. Metamodel modules are seen as templates out of which prototypical domain specific languages are composed.

Role-based Language Composition by Wende et al.

In [Wende et al., 2010], the idea of roles is used to leverage language engineering towards modularisation and composition. Complementing existing class types, the role construct is introduced as a kind of "place-holder" for other class types that may play a specific role. Thus, a role is treated as a kind of required interface of a language component. The composition operator is used to compose language components, which connects a role with a role player (class type) based on the semantics of subtyping. The approach extends the EMF and suggests subtyping semantics for implementing the composition operator, since it allows for information hiding between components. Unlike subclassing which is widely used for metamodel extensions, subtyping doesn't propagate class attributes and relationships to its subtype, but only expresses that objects of a subtype can be used wherever their supertype is expected (substitution semantics). The semantics of the composition (the glue logic) is defined in an external composition model. The target composite module is generated out of the composition model and the source modules using generative derivation by translating the composition operations to constructs known to EMF metamodelling language. Each composition model connects exactly two language components, one generic and one extended. A composition may consist of several role bindings (mappings), where each role binding maps a role to its role-playing class. Having the concept of role as a place-holder and the subtyping relationship for composition, this approach uses *parameterisation* to realise the composition. However, on the implementation level, the subtyping operator is realised using class *inheritance*.

Metamodel Mappings and Rules by Zivkovic et al.

In [Zivkovic et al., 2007] (see also Section 2.2), a two-step metamodel integration approach based on *metamodel mappings* and *integration rules* is proposed which integrates two metamodels into a new combined one. Semantically rich mappings are used to define structural and semantic correspondences between metamodel elements such as equivalence, containment, generalisation, etc. The semantics of the mappings are operationalised in a set of well-defined *invasive and non-invasive* integration rules which capture recurring composition operations. Since the mapping information is stored externally, it may be used to implement transformation rules for model migration, in case of consistency-breaking metamodel composition operations. Having an explicit external mapping language for composition, this approach is suitable for *generative, transformational derivation* to compose languages. There are no explicit constructs to support modularisation. Metamodels are treated as white-boxes. Supported non-invasive composition operators are inheritance and aggregation. An invasive merge operator is provided as well. Specific to this approach are metamodel mappings that may be defined even between structurally non-compatible elements. For example, if there is a semantic correlation between a concept modelled as an attribute and the other one as a class, an appropriate mapping may be defined between these two. A corresponding integration/composition rule will resolve the conflict invasively during the composition. Although such operations invasively modify the structure of the source metamodels usually breaking the consistency, such composition operations are useful to facilitate prototypical development of new hybrid metamodels.

Partial Domain Specific Models by Warmer and Kleppe

In [Warmer and Kleppe, 2006] metamodel element name-based references are proposed to combine partial textual DSLs. The approach introduces a meta type *reference to model element* which can be subclassed for a particular metamodel element which needs to be referenced. Instead of creating a hard-wired association to an element from another metamodel, a metamodel element from another metamodel is represented as reference to which source metamodel elements can refer to. The gluing is based on lightweight name-based matching. This composition operation may be categorised as a lightweight name-based importing operation. In addition, an inter-DSM validation check is provided, which checks whether the references between modules exist. This approach was introduced to support partial, modular development of domain-specific models and as an extension of Microsoft DSL Tools.

Viewpoint Unification by Vallecillo

In [Vallecillo, 2010] a s.c. view-point unification approach for combining domain specific languages is envisioned. Given the set of metamodels (viewpoints) to be combined and the set of correspondences between them, a new combined language is a unification of the single viewpoints with a set of functions (projections) that respect the constraints of correspondences. Projections are directed relations from the unified metamodel to single metamodels. As the author suggests, this approach proposal seems to be an umbrella approach for various different approaches as mappings between the languages may represent various composition operations such as inheritance, template instantiation, merge, etc. The actual derivation of the combined language seems to be transformational. However, it is not clear whether this is an automatic or a manual process of defining a new combined metamodel and how the projections to source metamodels are created. Furthermore, the approach doesn't address the modularisation aspect.

Integration of DSLs Using Ontological Foundations by Bräuer and Lochmann

In [Bräuer and Lochmann, 2007,Lochmann and Hessellund, 2009], an ontology-based approach is used to combine DSLs. While languages to be combined are lifted/mapped to a so called pivot ontology (semantic connector) that conforms to a dedicated upper ontology, the connections between language elements are specified using ontology object properties representing semantic *mappings* such as part-of, dependency, represents, etc. Similar to the approach by Vallecillo [Vallecillo, 2010], each participating DSL as an ontology must provide mappings/projections to the joined ontology. The use of semantic technology leverages the language integration by providing automatic reasoning and consistency checking of the unified ontology, that represents a unified view on all languages that are used in combination. This approach thus relies on a kind of mapping-based composition based on external links. However, the languages are used in coordination interchangeably, there is no composite language derived out of the combination. Since semantic links are external, they cannot be interpreted by the metamodelling tools. Additional tools must be provided and integrated to make use of the ontologies. The approach is suitable for coordinating existing modelling languages non-invasively, thus it doesn't focus on providing support for creating new metamodels and languages by modularisation and composition.

Model Weaving with Eclipse by Del Fabro et al.

Although not directly meant for language composition, in [Del Fabro et al., 2006], the authors propose to use generic model weaving to define corre-

spondences between models in general and metamodels in particular. The authors introduce a kind of an explicit, external model weaving language to model correspondences between (meta)model elements. The weaving metamodel introduces the concept of a weaving link (WLink), which enables the definition of links between model elements with simple link semantics. Moreover, the WLink can be extended (by inheritance), to be able to express different link types with specific and richer semantics. The set of defined links represents a weaving model. The links in a weaving model are non-invasive with respect to the referenced metamodels, as they just refer to the elements by relying on the name-based association. With respect to language composition, the authors, however, do not consider how a target combined metamodel may be produced based on the weaving model. Although meant to be a generic mapping information, the weaving model may be used also as an external glue logic for the composition. Since the mapping model is external (not part of the metamodelling language), it would be natural that a generative, transformational derivation could be applied, in order to generate a composite language. The authors do not cover the modularisation aspect of language composition neither. The model weaving approach has been combined with megamodelling[3] to solve the problem of the inter-DSL coordination considering inter-model traceability and navigability in model-driven engineering [Jouault et al., 2010].

Metamodel Templating by De Lara and Guerra

Apart from the mainstream metamodelling approaches, in [de Lara and Guerra, 2013], generic programming techniques such as *concepts*, *templates* and *mixin layers* are applied for metamodelling in order to increase the support for abstraction, modularity, reusability and extensibility of (meta)models and corresponding model management operations. Focusing on the usage of mixins, mixin layers rely on templating technique that allows for defining templated metamodel extensions (mixin layers), that can be applied on metamodels that conform to template parameters. The basic idea is to use the parameterised inheritance to realise a generic metamodel mixin. The "instantiation" of the template binds the mixin layer to a concrete metamodel that is the subject to extension and that conforms to the structure defined by the parameter type (concept). This kind of template instantiation may be categorised as dynamic invasive composition.

5.3.3 Evaluation of Approaches

A comparative evaluation of different approaches for metamodel modularisation and composition considering the classification framework introduced

[3]Megamodelling refers to the creation of s.c. mega-models [Bézivin et al., 2004], global models that assist in managing large collections of models.

before is illustrated in the Table 5.1. Each of the mentioned approaches has been evaluated according to the relevant modularisation and composition capabilities grouped into three main categories: metamodel modularisation, metamodel composition and composition technique.

- *Metamodel modularisation.* The majority of the approaches support some kind of white-box packaging of metamodels. The possibility to constrain access to metamodel elements by setting visibility modifiers, in order to achieve controlled white-boxes, was not found in any of the mentioned approaches. The package construct in MOF allows for flexible module structuring with module nesting possibilites, however without supporting the definition of explicit interfaces. On the other side, similar concepts such as the aspect in GME, the graphType in GOPPRR and the model type in ADOxx offer basic partitioning of metamodels into aspects. However, these concepts are primarily used as meta elements for defining various diagram types and less for language modularisation, thus without advanced concepts such as nesting and inter-metamodel reuse options. For example, one may not define a reusable module containing two diagram types that could be reused as a standalone module between arbitrary metamodels. Nevertheless, these concepts have been included into the evaluation as a limited form of modularisation. Furthermore, none of the approaches mentions the notion of grey-boxes. Finally, only two approaches suggest black-box encapsulation such as in [Weisemöller and Schürr, 2008] and in [Wende et al., 2010]. Both of them support the notion of explicit interfaces, however only on the level of classes.

- *Metamodel composition.* The compositional aspect has been evaluated according to the composition language characteristics such as nature, coverage and supported composition operators. Internal, native composition languages are part of the metamodelling language or serve as an extension of it, such as it is the case in MetaGME, MOF, GOPPRR and ADOxx. Other approaches propose explicit metamodel composition languages that finally transform composition directives to the natively supported constructs of the metamodelling language. Regarding the coverage, all approaches support usually the white-box composition of classes at some level. Some restrict descriptions of the approach solely to the concept of a class with a name, neglecting the inherent composition issues that arise when composing their structural features such as attributes and references. Each of the approaches supports a small subset of composition operators, having MOF as a metamodelling standard supporting the most among them. Inheritance is supported by all approaches. Importing operation is supported by the approaches that also support some kind of modular packages, as importing is a basic mechanism for reuse-by-reference between language

modules. Aggregation is supported inherently only by GOPPRR and ADOxx as these metamodelling languages allow for the reuse on the element atomic level. Template instantiation as a kind of reuse-by-copy operation is used in combination with parameterisation to instantiate elements into a new composite language. Stereotyping is used only in UML2. Annotation as defined in MOF for lightweight extensions is not used elsewhere. In addition to the standard composition operators, some approaches such as [Del Fabro et al., 2006] introduce the mapping operator as a part of an external mapping language, which requires application of external tools in order to be used for composition. Usually, such approaches use mappings for the s.c. coordination between existing metamodels and languages by allowing for model transformations and traceability between different languages. Furthermore, none of the approaches explicitly supports grey-box-based invasive composition, although, the approach proposed in [de Lara and Guerra, 2013] realises the idea of dynamic invasive composition. Finally, regarding the black-box composition, approaches such as [Weisemöller and Schürr, 2008] and [Wende et al., 2010] suggest external black-box parameterisation-based composition operators on the class level relying on the generative composition technique.

- *Composition technique.* It may be observed that both generative and interpretative approaches are equally present to derive new composite metamodels. However, a strong correlation exists between explicit composition languages and generative metamodel derivation, as one implies the other one. Furthermore, due to its transformational character, it becomes clear that the generative derivation is especially useful for the realisation of independent metamodel modules. The composite module is generated based on the copies of the source modules, which allows to define such modules as independent reuse blocks without planned, explicit dependencies to other modules. On the other side, the interpretative technique with explicit dependencies between modules requires rigorous design and planning of the modules. Furthermore, approaches which support importing operations based on reuse-by-reference are rather interpretative. On the other side, template instantiation and parameterisation are usually implemented using generative derivation as this implies reuse-by-copy of metamodel elements.

Table 5.1: Evaluation of metamodel modularisation and composition approaches

Feature	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Modularisation														
Encapsulation														
Packaging	+	+	+	-	-	+	+	+	-	-	-	-	-	-
Dependency	+	+	+	-	-	+	-	-	-	-	-	-	-	-
Reuse by reference	+	+	-	+	+	+	-	-	-	+	+	+	+	+
Reuse by copy	+	+	+	-	-	-	+	+	+	-	-	-	-	-
Information hiding														
White-box	+	+	-	+	+	+	-	-	+	+	+	+	+	+
Grey-box	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Black-box	-	-	+	-	-	-	-	+	-	-	-	-	-	-
Composition														
Nature														
Internal	+	+	+	+	+	+	-	-	-	+	-	-	-	+
External	-	-	-	-	-	-	+	+	+	-	+	+	+	-
Coverage														
Class	+	+	+	+	+	+	+	+	+	+	+	+	+	+
Attribute	-	-	-	+	+	-	-	-	+	-	-	-	-	+
Relation	-	+	-	+	+	-	-	-	+	-	-	-	-	+
Role	-	-	-	+	+	-	-	-	-	-	-	-	-	-
Model type	-	+	-	-	-	-	-	-	+	-	-	-	-	-
Composition operators														
Inheritance	+	+	-	+	+	+	+	+	+	+	-	-	-	+
Redefinition	-	+	-	-	-	-	-	-	-	-	-	-	-	-
Aggregation	-	-	-	+	+	-	-	-	+	-	-	-	-	-
Merging	-	+	-	-	-	-	+	-	+	-	+	-	-	-
Importing	+	+	-	+	-	+	-	-	-	-	-	-	+	-
Templating	+	-	+	-	-	-	+	-	-	-	-	-	-	+
Stereotyping	-	+	-	-	-	-	-	-	-	-	+	-	-	-
Annotation	-	+	-	-	-	-	-	-	-	-	-	-	-	-
Parameterisation	-	-	+	-	-	-	+	+	-	-	-	-	-	-
Composition technique														
Generative	-	-	+	-	-	-	+	+	+	-	-	n/a	+	-
Interpretative	+	+	-	+	+	+	-	-	-	+	+	n/a	-	+

[1] - Metamodel Modularisation and Composition by Ledeczi et al., [2] - Metamodel Modularisation and Composition in MOF and UML, [3] - Metamodel Components and Composition in MOF 2.0 by Weisemöller and Shürr, [4] - Metamodel Modularisation and Composition in GOPPRR, [5] - Metamodel Modularisation and Composition in ADOxx Meta^2-Model, [6] - Metamodel Modularisation and Composition in EMF Ecore, [7] - Metamodel Parameterisation by Pedro et al., [8] - Role-based Language Composition by Wende et al., [9] - Metamodel Mappings and Rules by Zivkovic et al., [10] - Partial Domain Specific Models by Warmer and Kleppe, [11] - Viewpoint Unification by Vallecillo, [12] - Integration of DSLs Using Ontological Foundations by Bräuer and Lochmann, [13] - Model Weaving with Eclipse by Del Fabro et al., [14] - Metamodel Templating by De Lara and Guerra

5.4 Chapter Summary

In this chapter, we provided basic concepts and an overview of the existing work on metamodel modularisation and composition in the context of metamodel-based modelling language engineering. In the first part, we introduced the basic notions of modular systems in general, and of metamodel composition in particular. Three major elements are important for modular, compositional metamodel definition: 1) modularised metamodels in terms of metamodel modules, 2) metamodel composition concepts in terms of flexible, comprehensive composition operators, and 3) adequate composition technique being either generative or interpretative. In the second part, we provided a systematic and extensive survey of the related work on metamodel modularisation and composition. For that purpose, a classification framework has been defined. Regarding the metamodel modularisation, we concluded that, although concepts for the encapsulation of metamodels into packages i.e. metamodel modules exist, extensive concepts for information hiding in terms of explicit interfaces that would support definition of not only white-boxes but also of grey-box and black-box metamodel modules are missing. In terms of metamodel composition, we distinguished between internal and external operators and between invasive and non-invasive operators that may operate on white-box, grey-box and black-box elements. White-box non-invasive composition operators are well supported. Internal, native, non-invasive black-box composition operators do not exist. On the other side, although external black-box composition operators have been suggested, the coverage of such composition operators is restricted to named classes without attributes. Finally, it could be observed that invasive grey-box composition operators as known from the invasive software composition are rarely addressed in existing approaches. The only exceptions are the annotation operator in MOF, however with very limited application and the template mechanism by De Lara and Guerra. In summary, based on the identified deficiencies, we may conclude that further concepts in metamodel modularisation and composition are required, in order to support the vision of systematic, modular definition of metamodels.

Chapter 6

A Concept for Modular Metamodel Engineering

"From a purely formal point of view, there is nothing that could be done with components that could not be done without them."

Clemens Szyperski

In this chapter, a concept for *modular metamodel engineering (MME)* is introduced as part of the main contribution of the underlying work. The concept may be seen as a continuation of the fragment-based method integration idea proposed by Kühn [Kühn, 2004]. Focusing on the language part of the method, and in particular, on the metamodel as a pivotal element in language definition, MME provides a systematic formalism for the realisation of modular metamodels within metamodelling platforms. MME aims at extending the metamodelling concepts, i.e. the concepts of a metamodelling language, in order to allow for the modular metamodel definition. On the one side, it introduces concepts to systematically define reusable, self-contained metamodel fragments. On the other side, it extends metamodelling languages with a set of composition operators to holistically support white-box, grey-box and black-box composition. Finally, it follows the purely interpretative composite language derivation as a chosen composition technique to realise both invasive and non-invasive metamodel composition.

After introducing the key requirements and notions of modular metamodel definition in Section 6.1, we elaborate on the metamodel modularisation in Section 6.2 and on the metamodel composition in Section 6.3, as fundamental aspects of MME. We introduce the notion of a metamodel fragment having explicitly defined interfaces and explicit dependencies only, and define metamodel composition operators that work on such fragments to allow for their combination. Since this chapter introduces metamodel mod-

ularisation and metamodel composition concepts on the conceptual level, in Section 6.4 we provide a formalisation of the introduced concepts by defining the metamodel for MME. Section 6.5 summarises this chapter.

6.1 Foundations of Modular Metamodel Engineering

In this section, the fundamentals of MME are provided. First, the basic idea of a holistic modular approach for metamodel engineering is introduced. Afterwards, the requirements on modular metamodel definition are discussed. Requirements are derived based on the fundamental elements of composition systems and by reflecting on deficiencies of the existing approaches for metamodel composition as elaborated in Chapter 5.

6.1.1 A Holistic Modular Approach

Based on the commonalities and variabilities of various approaches to language composition in Chapter 5, we have seen that only few if any of the approaches deal with a holistic view on metamodel composition. While some of them put emphasis on particular types of modularisation, the others focus on a specific composition operator(s). Instead, a holistic approach to modular metamodel definition should provide a solid ground for a multitude of modularisation and composition techniques, that arise from the combination of a variability of fragment types and composition operators. Here, we particularly mean that although the black-box metamodel composition may, by far, be the most flexible approach to define, manage and combine metamodel fragments, allowing for white-box and grey-box composition may equally be important in metamodel engineering projects. Having metamodel fragments only as black-boxes could easily lead to over-engineering or over-componentisation of metamodelling solutions, that could affect both the overall productivity of metamodel engineers and the performance of the underlying system. On the other side, white-box composition may complement the black-box composition in terms of fine-grained composition operators that contribute to reuse during the implementation of fragments. Finally, grey-box composition may allow for controlled, invasive modification of existing fragments where explicit extension points are not sufficient or the explicit modification, customisation or replacement of a metamodel fragment is not possible. Figure 6.1 illustrates the idea of extending the core metamodelling capabilities towards more holistic support for modular metamodel definition.

Each of the building blocks that extends metamodelling in order to enable MME is subject to detailed discussion in the subsequent sections.

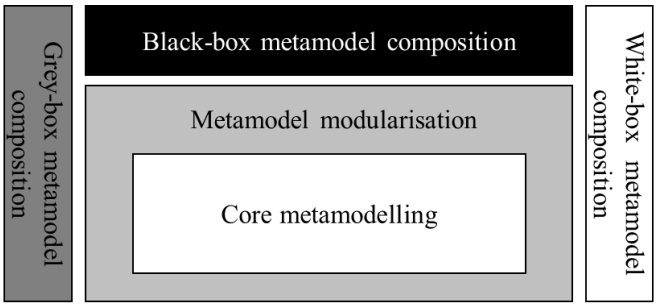


Figure 6.1: Extension of metamodelling towards modular metamodel engineering

6.1.2 Requirements on a Modular Approach

In the following, we postulate the requirements on modular metamodel definition. The requirements cover three major elements of metamodel composition as introduced in Chapter 5, such as the modularisation language, the composition language and the composition technique.

Modularisation of Metamodels

Modularisation is about encapsulation and information hiding. In metamodel engineering, modularisation is a process of encapsulating metamodel elements into reusable blocks called fragments, in a way that allows for information hiding. Therefore, two basic modularisation requirements (RMs) may be derived:

- *RM1 (Encapsulation).* It should be possible to *package* fragments of metamodels, i.e. a bundle of metamodel elements into self-contained, *reusable* metamodel fragments with clear fragment ownerships and dependencies. Such fragments should allow for packaging of core metamodelling constructs on both atomic and compound level.
- *RM2 (Information hiding).* It should be possible to explicitly hide the internals of a metamodel fragment from the view outside of the fragment by means of *explicit and implicit interfaces* and *declared visibilities*. Therefore, it should be possible to define different types of fragments such as black-boxes, grey-boxes and white-boxes.

Composition of Metamodels

Metamodel composition is about combining metamodel elements into composite metamodel structures. Having metamodel elements contained in different fragments, metamodel composition allows for metamodel fragment combination. Following requirements (RCs) on metamodel composition may be defined:

- *RC1 (Extensive coverage).* Metamodel composition must deal not only with trivial attributeless classes, but must also cover other core metaclasses of a metamodelling language such as classes, relations, endpoints, attributes and model types.

- *RC2 (White-box, grey-box and black-box composition).* On the level of fragments, metamodel composition should provide composition operators that may flexibly combine different types of fragments such as white-boxes, grey-boxes and black-boxes.

- *RC3 (Non-invasive and invasive composition).* Metamodel composition operators must allow for the flexible non-invasive composition of black-box fragments based on explicit interfaces, but also for the invasive composition of grey-box and white-box components. It should be possible to compose base fragments at their interfaces and extension points without a need to syntactically modify them.

Metamodel Composition Technique

The composition technique describes how the metamodel composition system is realised. Here, we focus on two aspects, the realisation of composition language and the derivation technique. The following requirements (RTs) can be identified:

- *RT1 (Internal, native language support).* Both the language constructs for specifying metamodel fragments and for the composition definition should be *natively supported* by the metamodelling language, i.e. the composition language nature should be *internal*. This requirement should allow language engineers to seamlessly adopt the modular metamodel definition without a need to learn completely new languages.

- *RT2 (Interpretative composition derivation).* While having a native, internal language support for composition, the composition system should derive new composite structures by interpretation instead of transformation. The composite fragments must be conflict-free, syntactically well-formed design artefacts.

- *RT3 (Compatibility with model-level mechanisms).* The compatibility with model-level mechanisms must be ensured. Metamodel definitions constructed in a modular way shouldn't be different to those designed using conventional techniques. This requirement is a linguistic problem, which should prove the assumption that modularisation and composition aspects contribute solely to the supporting capabilities of metamodelling languages (see discussion in Section 4.2.2).

6.2 Modularisation in Metamodel Engineering

To paraphrase Szyperski [Szyperski, 2002], there is nothing that can be done with components, that cannot be done without them. Clearly, components do not influence result of the end product, but the way how we develop the end product. While some metamodel may be a huge monolithic, complex, incomprehensible design artefact, the same result may be achieved, in a more productive and flexible way, through the composition of clearly separated, self-contained, metamodel fragments. Thus, the ability to define metamodel fragments is a basic requirement for modular definition of metamodels. According to Szyperski [Szyperski, 2002][1], in component-based software engineering, a software component is defined as follows:

> "A software component is a *unit of composition* with contractually specified *interfaces* and *explicit context dependencies* only. A software component can be *deployed independently* and is *subject to composition* by third parties."

This definition introduces important properties of components. An explicit interface is a provided interface of a component, whereas explicit context dependencies are required interfaces of other components (dependencies, since the interfaces are imported from other components). Another key characteristic is the independent deployment of components. A component is self-contained, and does not have any dependencies other than those explicitly declared. Finally, as a unit of composition, a component is used by other components over interfaces. In the following, those and other key characteristics of components in the context of metamodel fragments are introduced.

6.2.1 Metamodel Fragment

Derived from the definition of a software component, a metamodel fragment is defined as follows [Živković and Karagiannis, 2015]:

> "A metamodel fragment is a *unit of composition* with contractually specified *provided interfaces* and *required interfaces* only. A metamodel fragment can be *deployed independently* and is *subject to composition* by third parties."

A metamodel fragment is a packaging construct that combines a set of metamodel elements into a bundle. The fragment elements represents either fragment implementation or fragment interface definition. The implementation of the fragment defines the internals of the fragment, i.e. what the

[1] In the same book, several other definitions of a software component by various other authors are listed and comparatively analysed.

fragment does, whereas the interfaces define how a fragment may be used, without knowing its inner structure, i.e. how the fragment is implemented. This way, a fragment supports both encapsulation and information hiding.

- *Fragment implementation.* The internal structure of a fragment (fragment implementation) consists of *core metamodel elements* such as model types, classes, attributes, relations, etc. In order to modularise and organise internal structure of a fragment, a fragment may nest other fragments and build nested hierarchies. Elements of a nested fragment are implicitly and transitively owned by a nesting fragment. An *atomic* fragment is a fragment that doesn't contain any nested fragments but only atomic elements. A *composite* fragment consists of at least one other nested fragment. Direct fragment elements may be available to the outside via interfaces or by controlling their visibility using access modifiers. Access modifiers such as public, private, protected, etc, may be used to explicitly declare accessibility of a fragment element. Note, for a pure black-box fragment, elements are accessible only via explicit interfaces.

- *Fragment interfaces.* An interface defines a contract between a fragment that provides a certain structure and a fragment that uses that structure. A metamodel fragment interface describes a subset of a metamodel structure that is available for the use by other fragments (provided). A metamodel fragment may also explicitly list the interfaces it depends on (required interfaces). We further explain interfaces in the next section.

- *Explicit dependencies.* A fragment may depend on other fragments. Explicit dependencies allow for further accessibility and reuse of any kind of elements (interface or accessible inner element) of another fragment and is a prerequisite for building composite fragments in the course of metamodel composition.

Figure 6.2 illustrates the notion of a metamodel fragment.

6.2.2 Explicit Interfaces (Black-Box)

A fragment may expose a set of interfaces, in order to hide its internal implementation [Živković and Karagiannis, 2015]. The interface concept, which enables information hiding, is the cornerstone of a black-box metamodel composition. A metamodel fragment that exposes explicit interfaces is a black-box composition unit that can be combined by other fragments in various contexts. The interface concept allows for the loose coupling of fragments, which, in turn, promotes replacement or internal modifications

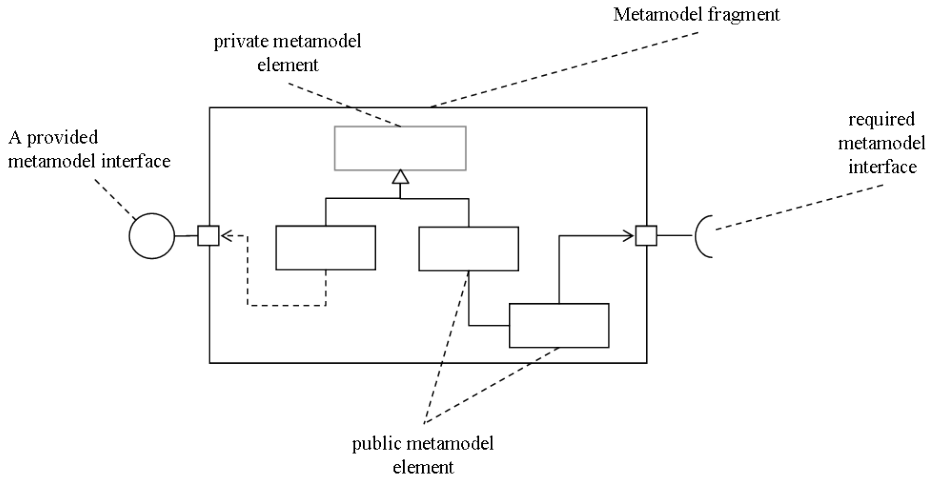


Figure 6.2: The notion of a metamodel fragment

while keeping interfaces stable. Interfaces may either be *owned* by a fragment or *imported* from another fragment. A metamodel fragment may have provided and required interfaces.

- *Provided interface.* A *provided interface* of a fragment exposes a subset of the internal metamodel implementation to other fragments. A provided interface is an interface that a metamodel fragment implements/realises. Figure 6.3 illustrates the notion of the provided interface. For example, the metamodel fragment *Process Model Metamodel* exposes the interface *ITask*, which is implemented by the internal class element *Activity*. In turn, another fragment that implements a language for creating business models, the *Business Model Metamodel* imports and uses this interface to state that the *Business Actor is responsible* for a certain task. The main advantage of having an explicit provided interface for the fragment *Business Process Metamodel* is its independent deployment and flexibility of change. Without knowing who and how will use it, the fragment may be independently reused and deployed as a unit of composition. Any internal change on the Activity class will not influence other fragments, as long the Activity class conforms to its interface.

- *Required interface.* A required interface of a metamodel fragment specifies explicit context dependencies to other fragments. A required interface is, therefore, always realised outside of a fragment, by a third fragment. The fact that a required interface is owned but not implemented by an owning fragment promotes the definition of predefined and explicit fragment *extension points*. A required interface that is owned specifies a contract to other fragments that may provide reali-

sations of that interface. An abstract fragment is a fragment owning at least one required interface. In Figure 6.3, the fragment *Business Process Metamodel* defines an extension point, the required interface *IPerformer* for the concept of a task performer. Any fragment may thus implement this interface to represent an actual performer of a task. In our case, this is the class *Role* from the fragment *Organisation Metamodel*.

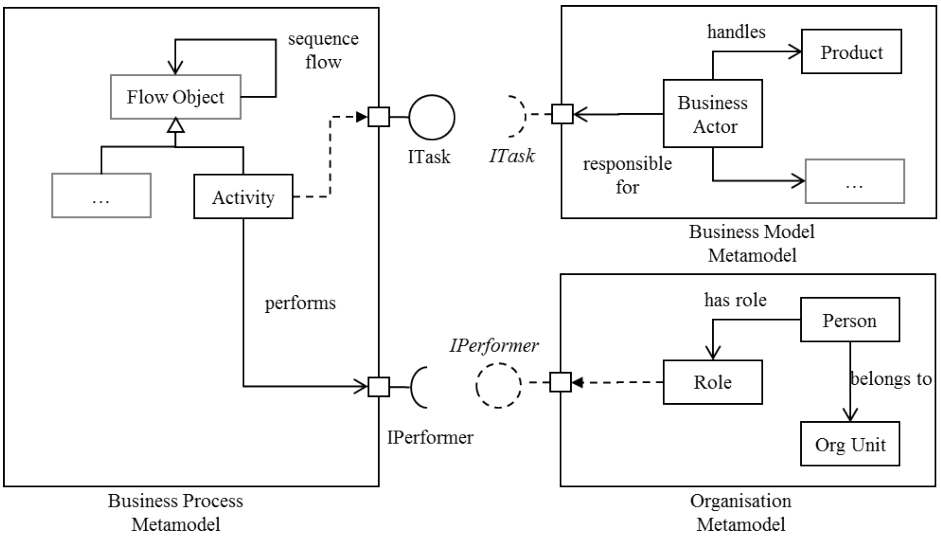


Figure 6.3: Provided and required interfaces. The example prototypically modularises the otherwise monolithic E-BPMS-Framework metamodel as defined in [Kühn et al., 2001]. The UML2 lollipop and socket notation is used to depict provided and required interfaces, respectively. A dashed line represents an imported interface, a full line is an owned interface.

6.2.3 Implicit Interfaces (Grey-Box)

The idea of implicit interfaces to realise grey-box metamodel composition is based on the notions of invasive software composition for program code as introduced in [Aßmann, 2003]. The basic idea behind invasive software composition relies on having code fragments such as fields and methods that may be injected into an existing program code such as class definitions based on implicit extension points[2][3]. The concept of implicit interfaces may be

[2]The notion of independent code fragments that describe state (fields) and behaviour (methods) called subjects, which may be integrated into classes representing objects is the cornerstone of the subject-oriented programming and has been initially introduced by [Harrison and Ossher, 1993]

[3]A similar idea to extend existing types is realised in C#, in which so-called *Extension Methods* allow for injecting methods to an existing base type without creating a new derived type, recompiling, or otherwise modifying the base type [MSDN, 2015].

applied on metamodels, in order to gain additional variation points where metamodel fragments may be extended [Živković and Karagiannis, 2016]. According to [Živković and Karagiannis, 2016], *an implicit interface represents an extension point of a metamodel element, which is implicitly defined by the inherent semantics of the underlying metamodelling language.* Each metaclass of a metamodelling language may have different implicit interfaces. For example, as depicted in Figure 6.4, a meta element *Class* may have as an implicit interface an *extensible set of member attributes.* Another implicit interface may also be a *set of explicit interfaces*, that class implements. Using appropriate composition operators, another fragment may access implicit interface and extend that particular class by injecting additional member attributes, or further supported interfaces. Implicit interfaces are crucial in metamodel composition scenarios, where extensional composition should take place on previously not explicitly defined extension points, or on non-modifiable elements.

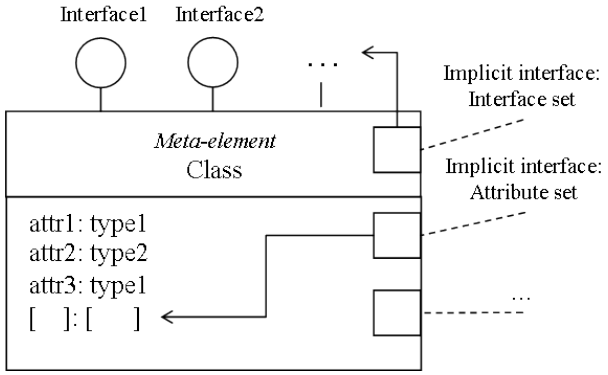


Figure 6.4: The notion of an implicit interface based on the metamodel element type Class

6.2.4 Explicit Access Modifiers (Grey-Box, White-Box)

In contrast to the idea of explicit interfaces and black-box encapsulation of metamodel fragments, explicit access modifiers are used to declare directly the accessibility of metamodel elements. By default, all elements are public, i.e. all are accessible for composition. However, some elements in metamodel fragments may be declared private to that fragment. In Figure 6.2 some elements of a metamodel fragment are declared as private, the others as public. Explicit access modifiers contribute to both grey-box and white-box metamodel composition, by allowing for a certain level of information hiding. Access modifiers are known language constructs in GPLs such as C++, Java or C#. Access modifiers differ in their accessibility level and domain.

- *Accessibility levels.* Metamodel elements may have different accessibility levels from the least to the most restrictive ones. Common modifiers for all GPLs are public, protected and private. A *public element* is globally accessible within and outside of its container. A *protected element* is accessible for its direct container and its derived elements. Protected is applicable for members of hierarchical elements such as attribute accessibility within classes. A *private element* is visible only within its container element. Different metaclasses may have different default accessibility levels for their members. For example, in a black-box fragment, default access to concrete elements may be private, whereas for interfaces it is always public.

- *Accessibility domain.* Accessibility domain of a metamodel element is defined by accessibility level and domain of a metamodel element in which that element is directly contained. The accessibility domain of a contained element can never exceed that of its container domain. For example, if a private class *C1* in a metamodel *M1* contains a public attribute *a*, that attribute will not be accessible from within another metamodel *M2*, even though the attribute accessibility level is public.

6.3 Composition in Metamodel Engineering

Modular metamodel definition is about the composition of metamodel fragments. Based on the modularisation type (black-box, grey-box, or white-box) of metamodel fragments to be combined, three different types of metamodel composition may be distinguished. Composition of black-box metamodel fragments with *explicit interfaces* implies *black-box metamodel composition.* Combining metamodel fragments based on *implicit interfaces* is what is called a *grey-box metamodel composition.* Likewise, *white-box metamodel composition* is applicable on accessible elements of white-box fragments. Figure 6.5 illustrates next to each other these three composition approaches to explicate the variability which arises based on the *"shades of grey"* of the metamodel modularisation. For each of the composition approaches, appropriate metamodel composition operators are needed. In the following, these three metamodel composition types are subject to detailed discussion.

6.3.1 Interface-based Black-Box Metamodel Composition

The black-box metamodel composition combines black-box fragments based on explicitly defined interfaces. The actual composition, however, requires the existence of appropriate, flexible composition operators. For the purpose of black-box composition, we introduce two new interface-based metamodel composition operators: *interface realisation* and *interface subtyping* [Živković and Karagiannis, 2015].

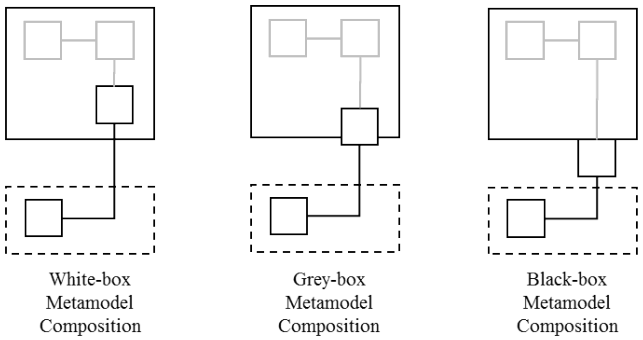


Figure 6.5: White-box vs. grey-box vs. black-box metamodel composition

- *Interface realisation.* Interface realisation binds one internal, concrete metamodel element to an interface element, thereby capturing the realisation of an interface by a concrete element. The realisation is valid only if an element conforms to that interface by contract. A concrete metamodel element may provide realisations to an arbitrary number of interfaces. Likewise, a single interface may have realisations from an arbitrary number of concrete elements. Interface realisation is a fundamental concept of a black-box metamodel definition. It promotes an explicit and controlled exposure of metamodel elements via provided interfaces and the realisation of required interfaces of a fragment. The operator is non-invasive as only the composer element, i.e. the concrete metamodel element is modified whereas the base element, i.e. the interface, remains unchanged. Figure 6.6 illustrates an example of usage of the interface realisation composition operator. Given the interface *ISimulatableTask*, that requires that each implementing element contains two attributes needed for the simulation of activities *Time* and *Costs*, the class *Task* is said to realise the interface *ISimulatableTask* by providing the required attributes *Time* and *Costs*. Interface realisation combines these two elements in a way that the class *Task* represents a valid implementation of the interface *ISimulatableTask*.

- *Interface subtyping.* Interface subtyping allows for extensions of interface definitions. It is a reliation between a base interface and a derived sub-interface with substitutability semantics. A subtype interface extends a base interface by providing additional members. The subtyping relation cardinality is unconstrained on both sides, such that an interface may extend and be extended by many other interfaces. The interface subtyping operator is non-invasive, since the subtype interface acts as a composer, whereas the base subtype remains unmodified. This is similar to the inheritance, as well as the fact that subtyping allows the reuse of structural features of interfaces along the

interface hierarchy. Due to its extensional and substitutability semantics, interface subtyping contributes significantly to the flexibility of metamodel composition scenarios. Figure 6.7 illustrates a concrete example of the subtyping usage. Let us suppose we define a metamodel fragment that contains some basic abstract metamodel concepts such the interface *INamedElement*, which in turn requires that each realising element must contain the attribute *Name* in its definition. Such fragment can be reused by many other fragments in their interface definitions, whenever the attribute *Name* is required. For example, the interface *ITask* subtypes the interface *INamedElement* in order to inherit the supertype interface specification.

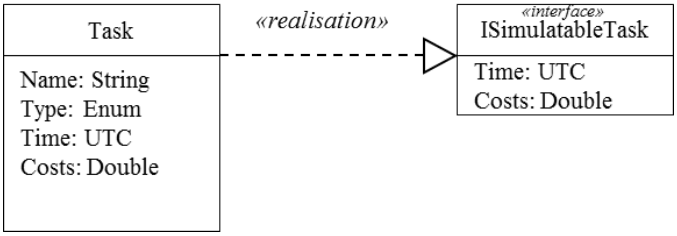


Figure 6.6: The notion of the interface realisation composition operator

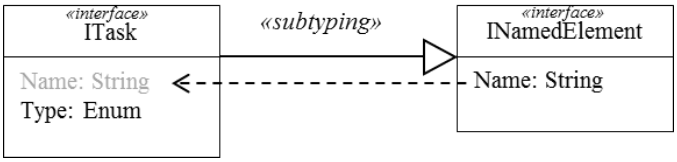


Figure 6.7: The notion of the interface subtyping composition operator

6.3.2 Extension-based Grey-Box Metamodel Composition

The grey-box metamodel composition relies on the notion of implicit interfaces to combine metamodel fragments into new composites. To compose fragments, we introduce the *extension composition operator* that allows to add features to an existing element without a need to modify it [Živković and Karagiannis, 2016]. The extension mitigates the deficiencies of the inheritance in that it doesn't require the introduction of a new derived type in order to extend the base type. Introduction of the derived type may in some composition and customisation scenarios be undesirable or not possible (the base class is already in use, i.e. instances exist that would require tool recompilation and model migration). Instead, the extension operator injects the extensions into the base type without modifying it. To combine

elements based on extension, a *base element*, an *extender element* and an *extension composition operator* are required.

- *Base element with implicit interfaces.* A base element may be any compound metamodel element, for which at least one implicit interface exists. Furthermore, a base element may be any metamodel element that is not atomic. For example, it doesn't make sense to extend elements such as attribute types that do not aggregate other elements and features. Element types such as classes, relations or model types are adequate extensional elements, since they contain implicit interfaces or extension points, given by the inherent semantics of the underlying metamodelling language. For example, as previously mentioned, a set of attributes is a natural extension point of any attributable element type.

- *Extender element.* Extender element is a kind of a *wrapper*, that defines the concrete extensions that should be injected to the base element. Since it is a pure utility construct, it is a non-instantiable, abstract element. In addition, the extender element must be of the same meta type as the base element. This is required to implicitly constrain only extensions that are possible for that specific element type.

- *Extension composition operator.* Extension operator is a relation that takes a base element and an extender element as input and extends the base element by injecting extensions based on well-defined implicit interfaces. For example, a class C_1 with an attribute a_1 may be extended by an attribute a_2 of an abstract class C_2 by declaring that the abstract class C_2 extends the class C_1. Like in inheritance, but inversely, the features of the extender element are propagated to the base element without any syntactic modification of the base element. An extender element may extend many base elements. In turn, a base element may be extended by arbitrary extender elements. The extension composition operator is invasive as the composer dynamically modifies the base element. One can think of the extension composition operator as a kind of a reverse-inheritance mechanism, where a derived element has a knowledge, i.e. 'decides' which base classes it extends. Figure 6.8 illustrates a concrete example. The class *Task* represents the base class that should be extended with two attributes *Time* and *Costs* however without any syntactic modification of that class. On the composer side, the class *SimulationActivityExtender* represents the extender element which contains those two attributes. By applying the extension operator from the extender class on the base class, the class *Task* dynamically receives all structural features of the class *SimulationActivityExtender*, i.e. the attributes *Time* and *Costs*.

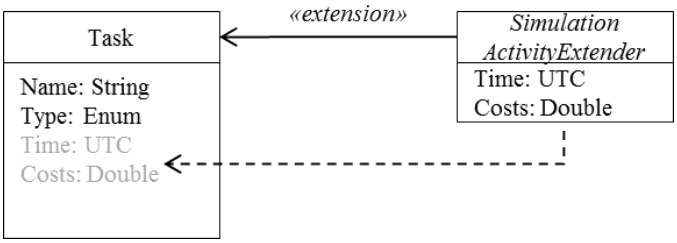


Figure 6.8: The notion of the extension composition operator

6.3.3 Mixin-based White-Box Metamodel Composition

The white-box metamodel composition combines accessible metamodel elements of metamodel fragments based on appropriate composition operators. White-box fragments expose metamodel elements directly without using interfaces, thus allowing for creating explicit dependencies of elements between fragments. However, if done in a systematic way, white-box composition contributes to the systematic decomposition of metamodels into modularised implementation fragments. For example, a black-box metamodel fragment that exposes interfaces, may include several nested white-box fragments that contribute to the implementation of the provided interfaces. By recombining those implementation fragments, new composite implementations may be developed in a productive modular way fostering reuse. Since the composition takes place on concrete metamodel elements, standard composition operators such as merge, inheritance, aggregation, etc. may be used as defined in Section 5.2. In addition to the standard operators such as inheritance, further composition operators are sought that allow for an increased reuse of metamodel fragments. Such composition operators should overcome issues that arise from the use of inheritance and should be used as alternative approaches to multiple inheritance. One such composition operator is introduced that is based on the idea of *class mixins*. Adopting the general idea of mixin-based inheritance [Bracha and Cook, 1990] in programming languages[4] in the context of metamodelling languages and metamodel composition, mixins are said to allow for the definition of independent metamodel element parts (Mixins) that may be reused, i.e. *mixed*

[4]In the area of programming languages, the idea of mixins has been around for years. The term was coined in the language Flavors [Moon, 1986], however, mixins have been initially defined as a formal language construct for language CLOS [Bracha and Cook, 1990]. Mixins found usage in OOPLs such as Smalltalk [Bracha and Griswold, 1996], and Scala [Odersky et al., 2004]. GPLs such as C++, that do not support mixins natively, aim at emulating the behaviour of mixins based on parameterised inheritance and template classes [Smaragdakis and Batory, 2001]. Similarly, in [Ancona et al., 2000] an extension for Java has been proposed called *Jam*, to allow for mixin-based class composition.

by other elements[5] [Živković and Karagiannis, 2016]. Mixins usually bundle some common set of features that may be shared among other metamodel elements. To allow for mixin-based metamodel composition, a *parent element*, a *mixin metamodel element* and a *mixin inclusion composition operator* are needed.

- *Parent element.* A parent element in the mixin-based composition may be any element of a compound type (compound metaclass), i.e. an element that contains other elements. For instance, a class is a compound element that may contain elements such as properties and references.

- *Mixin element.* A mixin element (a flavour) is a compound element type that contains other elements to be shared among other elements. It must be a non-instantiable, abstract element to denote its partial implementation and its single purpose of being a wrapper of structural features. The mixin element must be of the same type as the parent element. For example, an abstract element of type Class may be defined that contains a set of common structural features (attributes and/or references) that may be shared between various other classes.

- *Mixin inclusion.* Mixin inclusion composition operator takes a parent element and a mixin element as an input, and includes ("mixes-in") the features of the mixin element to the parent element. Mixin inclusion is non-invasive, since the parent element acts as a composer, whereas the mixin remains unmodified as the base element of the composition function. Figure 6.9 illustrates the application of the mixin inclusion operator. The abstract mixin class *VersionableMixin* is defined that contains the string attribute *Version* and the enumeration attribute *State*. The parent class *Document* owns two attributes *Name* and *Type* but in addition it declares to *mixin* the class *VersionableMixin* in order to allow for document versioning and state-based workflow. By applying the mixin inclusion from the class *Document* to the class *VersionableMixin*, the class *Document* receives all structural features of the mixin class, in our case, the attributes *Version* and *State*. Note that the type restriction applies. Both elements are of the same metaclass type.

In what it does, mixin inclusion is similar to the aggregation. However, it doesn't aggregate single elements as structural features but a predefined set of elements bundled in a mixin element. In fact, mixin inclusion operator has semantics much closer to the inheritance, when it comes to the reuse of structural features of the base class. That said, mixin inclusion complements

[5]The term mixin is inspired by the s.c. ice cream Mix-In, an extra ice cream flavour that may be combined on top of a base ice cream.

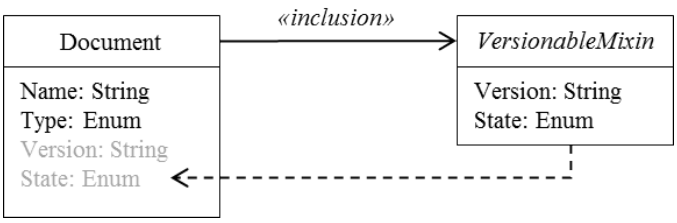


Figure 6.9: The notion of the mixin inclusion composition operator

single inheritance in cases where the reuse of structural features from multiple elements is desired, but due to the inheritance singleness is not possible. Even when multiple inheritance is allowed, the mixin inclusion has a clear advantage over multiple inheritance as being a lightweight operator. Since it restricts mixin elements to be abstract elements only (partial elements), the mixin operator avoids the creation of complex inheritance hierarchies, while still allowing for multiple inclusion of bundles of structural features defined at mixin elements.

6.4 A Metamodel for Modular Metamodel Engineering

In the previous sections we introduced the main concepts of modular metamodel engineering. In the following, we summarise these concepts in a *metamodel for modular metamodel engineering*. The metamodel represents a *conceptual framework* for MME. The central concept is a *metamodel fragment*. A fragment may *nest* other fragments. A fragment is a *composite* fragment if it nests other fragments, otherwise it is *atomic*. In addition, fragments may declare *dependencies* between each other. A fragment may existentially *own* metamodel elements or may *import* elements from other dependent fragments. In turn, a metamodel element may be owned only by one fragment, but may be imported i.e. reused by arbitrary number of other fragments. A metamodel element in a fragment may be a class, a model type, an attribute, a relation, etc. Furthermore, a *metamodel element* may represent a *concrete element* or be an *interface element*. A concrete element may *realise* many interfaces. In turn, an interface may be realised by arbitrary number of concrete elements. Thus, a fragment may contain internal concrete elements and a set of provided and required interfaces that are exposed externally to other fragments for black-box composition. Other fragments may import provided interfaces and use them or import required interfaces and provide an appropriate realisation. Besides the interface realisation composition operation, numerous other composition operators exist. Interfaces support *subtyping* such that interface specification may easily be extended.

To allow for the grey-box composition, the *extension* composition operator is applied. Concrete elements that extend other elements are called *extender elements*. A fundamental operation for white-box composition such as aggregation is applicable on both concrete elements and interface elements. On the other side, the inheritance, and the *mixin inclusion* operator are applicable on concrete elements, only. Concrete elements that are included, mixed in by other elements are called *mixin elements*.

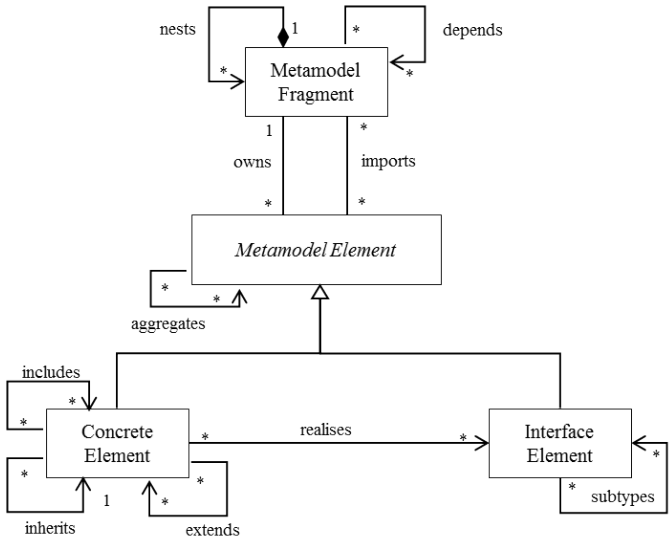


Figure 6.10: A metamodel for modular metamodel engineering

6.5 Chapter Summary

In this chapter, we introduced the main concept for modular definition of metamodels, as part of the general MME approach. MME extends existing metamodelling concepts in two ways. It contributes with concepts for the modularisation of metamodels into reusable, self-contained metamodel fragments with explicitly defined interfaces. On the other side, it offers a rich set of composition operators to flexibly combine such metamodel fragments. MME is holistic as it provides concepts not only for the black-box composition, but also for the grey-box and the advanced white-box metamodel composition. While black-box operators operate on the level of fragment interfaces, i.e. on explicit extension points of fragments, the grey-box composition allows for extending the fragments based on implicit interfaces. The grey-box composition complements the black-box composition in cases when explicit interfaces of fragments do not exist. The white-box composition operators contribute with advanced concepts to implement the internals of black-box fragments in a modular way, i.e. to combine implementation

fragments by fostering reuse. This chapter provided a conceptual overview of the MME approach, which serves as a conceptual framework for the specification of a language for modular metamodel definition and its realisation in metamodelling platforms. The formal specification and the realisation of the approach are the topics of the two subsequent chapters.

Chapter 7

A Language for Modular Metamodel Engineering (MMEL)

"The limits of my language are the limits of my mind. All I know is what I have words for."

Ludwig Wittgenstein

This chapter contributes with the formalisation of the concepts of the general MME approach introduced in the previous chapter by defining a *language for modular metamodel engineering (MMEL)*. This meta-language represents an extension of the existing metamodelling languages. It extends core metamodelling concepts with concepts for metamodel modularisation and metamodel composition. The MMEL consists of three basic modules. The core metamodelling language represents an abstract representation of the metamodelling concepts which are then extended by two other modules, the modularisation language and the composition language. The MMEL itself is formalised using the metamodelling techniques and corresponding means for syntax, semantics and notation definition.

The chapter is organised as follows. To begin with in Section 7.1, we first report on language specification formalisms used in the remainder of the chapter and introduce the basic language architecture. The upcoming sections are dedicated to three language modules. In Section 7.2 we specify the core abstract metamodelling language. Section 7.3 formalises the concepts of the metamodel modularisation language. In Section 7.4 the metamodel composition language is specified. Section 7.5 concludes the chapter.

7.1 Preliminaries

In this section, we shortly summarise the formalisms used to specify MMEL. Furthermore, we explain how the language is structured by introducing the general language architecture.

7.1.1 Note on Specification Formalism

In the following we report on formalisms used to define the syntax, semantics and notation of the MMEL.

Definition of the Syntax

For the definition of the language, and in particular, of the abstract syntax, we use metamodelling, i.e. a metamodel-based approach as a practical yet rigorous formalism to define the abstract syntax of a language[1] (see Section 3.3 for discussion on various approaches for language definition). In particular, we use the simplified UML class diagram notation based on the semantics of MOF [OMG, 2014] i.e. UML Infrastructure [OMG, 2011a], to define the elements of the abstract syntax, i.e. the *syntax schema* of the meta-language. We enrich the syntax schema definition by specifying the *constraints* using the natural language descriptions and by referring to the abstract syntax elements.

Definition of the Semantics

In order to specify the semantics of the introduced abstract syntax concepts, we use a mixture of a formal and informal semantics specification. We follow a kind of an axiomatic approach to define the *static semantics* of abstract syntax elements using precise natural language descriptions. The semantic axioms defined may however be expressed using, for example, first-order logic based languages for knowledge description such as OWL2 [Motik et al., 2009b]. Likewise, we identify and specify the *dynamic semantics* where appropriate by describing the behaviour of the system using the precise natural language. In addition, in the subsequent chapter that introduces language implementation, some of the semantic definitions will additionally be specified using operational language such as Java.

Definition of the Notation

Although the notation of MMEL is of less importance as different tools may have different concrete syntaxes for metamodelling, for documentation and visualisation purposes, we choose to specify the graphical syntax as

[1] Note that we actually use a metamodelling formalism to define the metamodelling language itself, i.e. to extend it.

one possible concrete syntax definition. Where possible, the introduced graphical syntax is based on the graphical UML notation. Where this was not possible, additional graphical notation elements have been defined (as stereotypes).

7.1.2 MMEL Language Architecture

MMEL has been architected based on well-established language design principles such as modularity, extensibility, layering, and reuse[2].

- *Modularity.* Modularity has been applied to decompose the language into meaningful self-contained logical units, i.e. packages, based on the principles such as separation of concerns, strong cohesion and loose coupling.
- *Layering.* Layering has been applied to separate concepts based on the abstraction level, such that higher-level abstract concepts can be used by the lower-level concrete concepts.
- *Extensibility.* Since the language is modular it is also open to extensions. The extensibility is systematically applied to flexibly extend the core metamodelling with modules for modularisation and composition. However, the language allows for adding other composition modules in the future.
- *Reuse.* Due to high modularity and layering, the core abstract concepts are reused to define the concrete core metamodelling concepts as well as to define modularisation concepts.

Figure 7.1 introduces the package structure of the MMEL. On the top level, the language consists of three main packages such as the core metamodelling language (CML), the metamodel modularisation language (MML), the metamodel composition language (MCL). The CML consists of two packages that encapsulate abstract and concrete metamodelling constructs. The MML extends the core metamodelling concepts from CML with two packages regarding encapsulation and interfacing constructs. The MCL depends on both CML and MML, and contributes with constructs for black-box, grey-box and white-box composition. In the subsequent sections, these language packages are subject to detailed elaboration.

7.2 Core Metamodelling Language (CML)

Metamodelling languages provide basic constructs for metamodel and language definition. In Chapter 3 a thorough overview of metamodelling lan-

[2]The same set of principles is used to define the UML language (compare [OMG, 2011a], p.11).

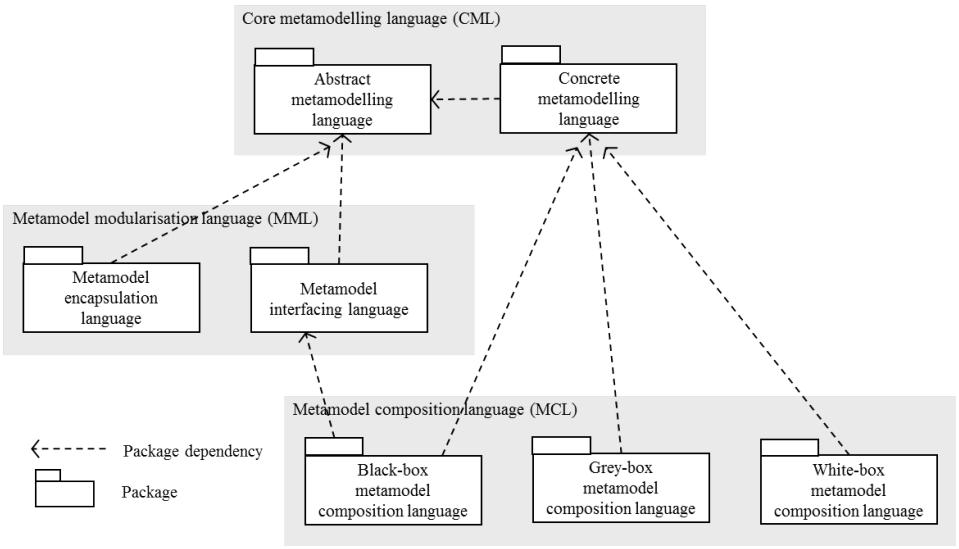


Figure 7.1: The package structure of MMEL

guage concepts as well as a comparative overview of existing metamodelling languages have been given. Based on this analysis, in this section a kind of common core metamodelling language is introduced, that abstracts from specific metamodelling implementations. Note that it is not our intent to define a new metamodelling formalism, but solely to have a common core meta-metamodel that provides basic constructs to define the "content" of the metamodel fragments. Such meta-metamodel is also not complete. However, it supports core capabilities such as basic and role constructs, as well as the basic intra-language reuse by inheritance and aggregation. All of these constructs cover the basic scenarios for defining internal elements of metamodel fragments.

We divide the core metamodelling language into the abstract part and the concrete part. The reason for such design is of the generalisation and the extensional nature. While the abstract language solely defines the basic abstract concepts and relations between those concepts without any concrete instantiable elements, the concrete language inherits the abstract structure and provides a concrete "implementation" of the metamodelling language. As we will see, the interface language is another language that will inherit the abstract structure of the core concepts, in order to allow for the seamless integration of the interface concepts.

7.2.1 Abstract Metamodelling Language

The abstract metamodelling language defines the structure of the common, core metamodelling language. We define the language by discussing its syntax and semantics. Since the language is abstract, the notation is not

given.

Syntax

Recalling the definition of the core and supporting capabilities of metamodelling languages (see Chapter 3), the common abstract metamodelling language should support all core capabilities, as well as the basic supporting capabilities for intra-language reuse such as aggregation and inheritance. In the following, the syntax schema and the constraints are subject to detailed discussion.

Syntax Schema The syntax of the core abstract metamodelling language consists of the following constructs that match the core metamodelling capabilities: *AbstractClass*, *AbstractAttribute*, *AbstractModelType*, *AbstractRelation*, *AbstractRelationEnd*. We explicitly prefix the element names with the *Abstract* keyword to denote their abstract, non-instantiable nature. Figure 7.2 illustrates the abstract syntax of the abstract metamodelling language.

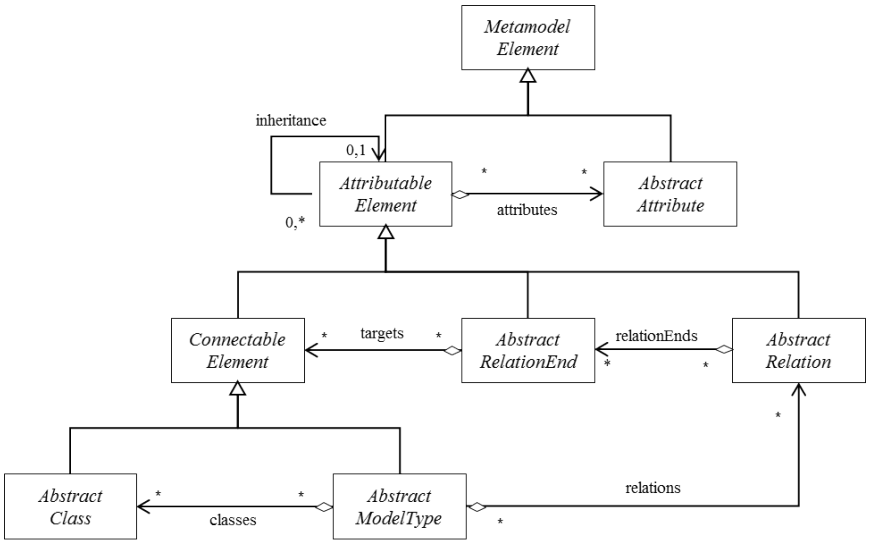


Figure 7.2: Metamodel of the abstract core metamodelling language

- *MetamodelElement.* A metamodel element represents the root element of the metamodelling language. It doesn't have any special semantics besides being the supertype of other metaclasses.
- *AttributableElement.* An attributable metamodel element is an abstract, non-instantiable construct representing a supertype of all metaclasses that may contain attributes. The metaclass *AttributableEle-*

ment supports single inheritance in order to enable reuse of attributes along the same type metamodel element hierarchy.

- *AbstractAttribute*. An abstract attribute represents a structural feature of a metamodel element and is of some attribute type, such as string, integer, double, bool, date, etc. Note, for the sake of simplicity, we do not consider attribute types any further.

- *AbstractClass*. An abstract class is a central metamodelling construct that is used to specify entities (classes) of a modelling language. Common to all approaches, a class may contain attributes and define class hierarchies using the inheritance. Hence, we specify that the metaclass *AbstractClass* inherits indirectly from the metaclass *AttributableElement* in order to support inheritance and attribute containment.

- *AbstractModelType*. An abstract model type is another attributable element and a metaclass used to typify models (diagram types). This construct is necessary to support hybrid modelling as it may be used to represent a conceptual boundary of one language concern. A model type is attributable and may support inheritance to reuse attributes. In addition, a model type contains classes and relations. Hence, we define first the metaclass *AbstractModelType* as an indirect subclass of the metaclass *AttributableElement*. Then, we define classes and relations to be two additional structural features of a model type by adding two aggregation relationships *classes* and *relations* that connect the metaclass *AbstractModelType* with metaclasses *AbstractClass*, *AbstractRelation*, respectively.

- *AbstractRelation*. An abstract relation defines relations between classes and/or model types. A relation is attributable and supports the inheritance of attributes. Relation does not connect to its targets directly, but over relation ends. The relation arity is determined by the number of relation ends. The arity should at least be binary, such that relation consists of exactly one *Source(From)* relation end and one *Target(To)* relation end. Nevertheless, to be generic, we leave the option to define n-ary relations. Thus, the metaclass *AbstractRelation* inherits from the metaclass *AttributableElement*. Furthermore, we define one aggregation relationship *relationEnds* to state that the metaclass *AbstractRelationEnd* is an additional structural feature of relations.

- *AbstractRelationEnd*. An abstract relation end specifies how a target of a relation end participates in a relation in terms of multiplicity and allowed target types. A relation end may directly support several target types by virtue of aggregation, or indirectly over the inheritance hierarchy of direct target types. Hence, the multiplicity for allowed targets is left unconstrained. Finally, a relation end is an attributable

element. Thus, the metaclass *AbstractRelationEnd* inherits from the metaclass *AttributableElement*. Furthermore, we define one aggregation relationship *targets* to state that the metaclass *AbstractRelationEnd* contains an additional structural feature, that of connectable elements.

- *ConnectableElement*. A connectable element represents an abstract generalisation construct of classes and model types with semantics of a targetable constructs of relation ends. Hence, we define the metaclass *ConnectableElement* as an abstract metaclass that is a superclass of metaclasses *AbstractClass* and *AbstractModelType*, and as a subclass of the *AttributableElement*.

It is worth considering that containment relationships between elements have the semantics of *weak aggregation*. Weak aggregation allows that elements may exist as standalone reuse artefacts. For example, classes may be contained by many model types based on the relationship *classes* that connects the metaclasses *AbtractClass* and *AbstractModelType*. The same is true for relationships *attributes*, *relations*, *relationEnds* and *targets*.

Constraints There are few additional constraints that further restrict the syntax of the abstract metamodelling language. Table 7.1 lists those constraints.

Table 7.1: Abstract syntax constraints of the core metamodelling language

#	Constraint	Description
C1	*Compatibility of inheritance*	It is not possible to define inheritance between different metaclasses. For example, a class cannot inherit from a model type, but only from another class.
C2	*Cyclic inheritance dependency*	An attributable element cannot inherit from itself, neither directly, nor indirectly.
C3	*Relation direction*	A relation must have at least one relation end of type From and one of type To.

Semantics

In the following, we specify the static and dynamic semantics of the abstract metamodelling language.

Static Semantics

Definition 1 (Inheritance) *Given the attributable elements AE_1 and AE_2 with corresponding sets of attributes Sa_1, Sa_2, AE_1 inherits from AE_2 only if for each of the attributes in Sa_1 the same attribute in Sa_2 exists.*

Definition 2 (Transitiveness of Inheritance) *Inheritance is a transitive relationship. Given the attributable elements AE_1, AE_2 and AE_3, if attributable element AE_1 inherits from AE_2 and AE_2 inherits from AE_3, then AE_1 inherits from AE_3.*

Definition 3 (Reflexiveness of Inheritance) *Inheritance is reflexive relationship. For any attributable element AE_1, AE_1 inherits from AE_1.*

Definition 4 (Derived Relation End Targets) *By virtue of inheritance, subelements of connectable elements that are relation end targets are also valid targets. Given the connectable elements CE_1 and CE_2, where CE_1 inherits from CE_2, and relation end RE_1, if CE_2 is a target of RE_1 then CE_1 is also a valid target of RE_1.*

Dynamic Semantics

Definition 5 (Inheritance of Attributes) *Given the attributable elements AE_1 and AE_2 with corresponding sets of attributes Sa_1, Sa_2, and the inheritance relationship R such that AE_1 inherits from AE_2, AE_1 aggregates all attributes from Sa_2 into Sa_1.*

Notation

Due to its abstract nature, the abstract metamodelling language does not feature any concrete notation.

7.2.2 Concrete Metamodelling Language

The concrete metamodelling language depends on the abstract language. It provides the "implementation" of the abstract constructs defined in the previous section. In the following, the syntax, semantics and notation are discussed.

Syntax

The concrete language introduces instantiable constructs for each core construct from the abstract language without adding any additional constraints.

Syntax Schema The syntax of the concrete metamodelling language consists of the following constructs, each of which inheriting from the corresponding metaclasses of the abstract metamodelling language: *Class*, *Attribute*, *ModelType*, *Relation*, *RelationEnd*. By virtue of class inheritance, each metaclass inherits structural features of its super metaclass. For example, the metaclass *ModelType* as a subclass of the metaclass *AbstractModelType* becomes automatically an attributable element and a container of classes and relations. Figure 7.3 illustrates the syntax schema of the concrete metamodelling language. No further constraints are added.

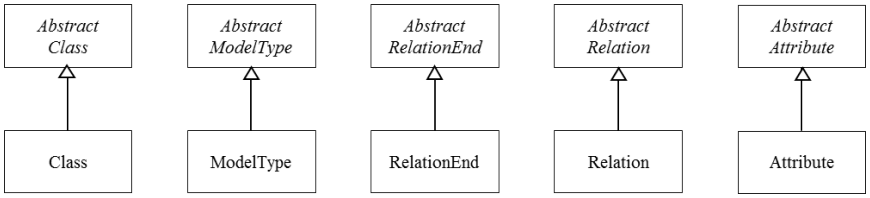


Figure 7.3: Metamodel of the concrete core metamodelling language. It extends the abstract counterpart.

Restrictions No additional restrictions are defined. Restrictions from the abstract language are inherited.

Semantics

The concrete language semantics is inherited from the abstract language. The only difference is that the concrete metaclasses are instantiable instead of being abstract.

Static Semantics No additional static semantic rules are defined. The static semantics of the abstract language apply.

Dynamic Semantics No additional dynamic semantic rules are defined. The dynamic semantics of the abstract language apply.

Notation

The concrete syntax of the core metamodelling language may be based on the existing UML class diagram notation [OMG, 2011b]. To differentiate between different metaclasses such as model type and class, the class symbol may be stereotyped. Figure 7.4 illustrates a simplified specification of a possible notation for the core metamodelling language. For each of the types, a class symbol with an according letter in the upper right corner is defined. Similarly, the core aggregation relationships are based on the

UML aggregation notation with an according letter to differentiate between different relationship types. In case of binary relations, having exactly one from and one to relation end and only one allowed target per relation end, an alternative convenient "syntactic sugar" is provided based on the UML association (including variants for composition, aggregation, navigable end, etc.). If any of the core types is abstract, this is annotated by the cursive font.

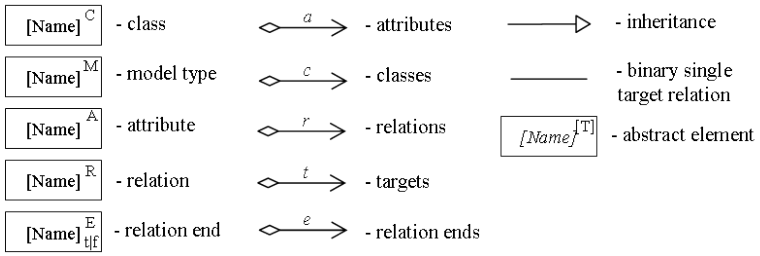


Figure 7.4: A concrete syntax of the core metamodelling language (Simplified specification)

7.3 Metamodel Modularisation Language (MML)

In this section, a metamodel modularisation language is introduced. It is defined as an extension of the core metamodelling language by adding constructs necessary to support modularisation of metamodels. In Chapter 6 we discussed the basic requirements and notions of metamodel modularisation. Based on it, the introduced language should provide appropriate concepts for both metamodel encapsulation (RM1) and information hiding (RM2) in form of black-box, grey-box and white-box metamodel fragments. For each of the modularisation aspects appropriate constructs are introduced packaged into sublanguages, *the metamodel encapsulation language* and *the metamodel interface definition language*, respectively.

7.3.1 Metamodel Encapsulation Language

The metamodel encapsulation language deals with the packaging of metamodel elements into reusable, modular units. It introduces metamodelling constructs such as a fragment and a containable element. In the following, the language definition is given divided into the syntax, the semantics and the notational part.

Syntax

We define the syntax of the metamodel encapsulation language by introducing the core syntax schema followed by a range of syntax constraints on top

of it.

Syntax Schema The encapsulation language introduces two basic constructs for decomposing the metamodels into reusable, modular fragments such as a *fragment* and a *metamodel element*.

- *Fragment*. A fragment is a fundamental modularisation construct. It allows for encapsulation of parts of metamodels as reusable modules. A fragment consists of metamodel elements, whereas a contained element in a fragment is either owned or imported. Elements that are owned by a fragment can only exist as parts of their owning fragments. Hence, there is an existential dependency between a fragment and an owned element. Imported elements are those referenced from other fragments. Imported elements realise the inter-fragment reuse of elements and provide the basis for many composition operators. Hence, we define the metaclass *Fragment* having two types of structural features, owned and imported metamodel elements. This is defined using relationships *ownedElements* and *importedElements*, respectively. Furthermore, as a prerequisite to import elements from other fragments, an explicit dependency to that fragment must be established. This is defined using the relationship *dependentFragments*. In addition, fragments may participate in nested structures, i.e. a fragment may contain other fragments, as its internal packages. We specify this feature using the relationship *nestedFragment*.

- *MetamodelElement*. The metaclass *MetamodelElement* is an abstract element imported from the abstract metamodelling language. From the point of view of encapsulation, it has semantics of a containable element. We will see that a containable element may be either an interface or a concrete implementation element, which is according to the idea of a fragment having internal implementation elements and explicit interface elements.

Figure 7.5 illustrates the modularisation extension of the core metametamodel that addresses the encapsulation constructs.

Constraints In Table 7.2, a list of additional constraints on syntax elements of the encapsulation metamodel is provided.

Semantics

The static semantics of the encapsulation language introduces precise semantics of abstract syntax elements. By doing so, some additional definitions of derived constructs are introduced such as top fragment, nested fragment, dependent fragment, that are not explicitly expressed through the abstract

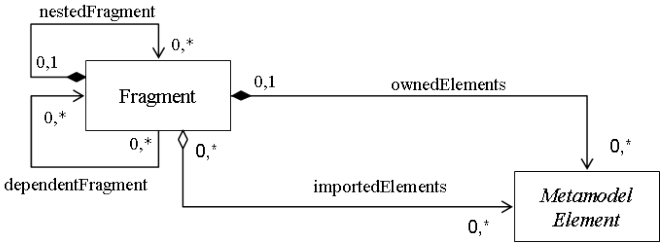


Figure 7.5: Metamodel of the metamodel modularisation language - Encapsulation module

Table 7.2: Abstract syntax constraints of the encapsulation part of the modularisation language

#	Constraint	Description
C4	*Acyclic fragment dependency*	A fragment cannot be dependent on itself, neither directly nor indirectly over other fragments.
C5	*Acyclic fragment nesting*	Likewise, a fragment cannot nest itself.
C6	*Fragment nesting*	Nested fragments may not be dependent fragments.
C7	*Fragment dependency*	Only top-level fragments, i.e. not nested fragments may form dependencies between each other.
C8	*Imported elements*	Only accessible elements of explicitly dependent fragments may be imported elements. This should also constrain that an owned element is also imported by the same fragment.
C9	*Recursive imports*	An already imported element cannot be subject to further imports.
C10	*Containment of member elements*	If a compound element is contained in a fragment, all of its member elements must be contained as well.

syntax. Dynamic semantics contributes to the behavioral meaning of the abstract syntax elements.

Static semantics

Definition 6 (Nested Fragment) *A nested fragment is a fragment that is nested by another fragment. Given the fragment F_1, its set of nested fragments Sn_1, and fragment F_2, F_2 is a nested fragment, if it is contained the set Sn_1.*

Definition 7 (Top Fragment) *A top fragment is a fragment that is not nested by any other fragment. Given the fragment F_1, its set of nested fragments Sn_1, and fragment F_2, F_2 is a top fragment, if it is not contained in the set Sn_1.*

Definition 8 (Atomic Fragment) *An atomic fragment is a fragment that doesn't nest any other fragments. Given the fragment F_1, its set of nested fragments Sn_1, F_1 is an atomic fragment, if its set Sn_1 is empty.*

Definition 9 (Compound Fragment) *A compound fragment is a fragment that nests at least one other fragment. Given the fragment F_1, its set of nested fragments Sn_1, F_1 is an atomic fragment, if its set Sn_1 is not empty.*

Definition 10 (Dependent Fragment) *A dependent fragment is a fragment that contains at least one dependency to other fragments. Given the fragment F_1, its set of dependent fragments Sd_1, F_1 is a dependent fragment, if its set Sd_1 is not empty.*

Definition 11 (Independent Fragment) *An independent fragment is a fragment that doesn't contain any dependencies to other fragments. Given the fragment F_1, its set of dependent fragments Sd_1, F_1 is an independent fragment, if its set Sd_1 is empty.*

Definition 12 (Owner Fragment) *An owner fragment is a fragment that owns at least one metamodel element. Given the fragment F_1, its set of owned metamodel elements So_1, F_1 is an owner fragment, if its set So_1 is not empty.*

Definition 13 (Owned Element) *An owned element is an element that is owned by an owner fragment. Given the fragment F_1, its set of owned metamodel elements So_1, and metamodel element ME_1, ME_1 is an owned element, if it is contained in the set So_1.*

Definition 14 (Imported Element) *An imported element is an element that is owned element of one fragment that is imported by another fragment. Given the fragment F_1, its set of dependent fragments Sd_1, its set of owned metamodel elements So_1, its set of imported metamodel elements*

Si_1, fragment F_2, its set of owned metamodel elements So_2, and metamodel element ME_1, where ME_1 is contained in So_2, not contained in So_1 and F_2 is contained in Sd_1, ME_1 is an imported element of F_1 if it is contained in Si_1.

Definition 15 (Nesting Transitivity) *Nesting of fragments is transitive. Given the three metamodel fragments F_1, F_2 and F_3, if F_1 nests F_2, and F_2 nests F_3, F_1 also nests F_3.*

Definition 16 (Nesting Reflexivity) *Nesting of fragments is reflexive. For any metamodel fragment F, F is implicitly a nested element of F.*

Definition 17 (Dependency Transitivity) *Fragment dependency is transitive. Given the three metamodel fragments F_1, F_2 and F_3, if F_1 is dependent on F_2, and F_2 is dependent on F_3, F_1 is also dependent on F_3.*

Dynamic Semantics

Definition 18 (Deletion of Nested Fragments) *If a compound fragment is deleted, all nested elements must also be deleted.*

Definition 19 (Deletion of Owned Elements) *If a fragment is deleted, all owned elements must also be deleted.*

Notation

The concrete syntax of the encapsulation aspect of the metamodel modularisation language may be based the graphical syntax of the UML2 package diagram as defined in [OMG, 2011b]. Figure 7.6 illustrates a possible concrete syntax. A metamodel fragment may be represented by a UML element package. Basically, the content of the fragment, nested fragments, owned and imported elements, may be displayed inside of the fragment symbol or alternatively using the relationships between packages and elements. Imported elements are represented with dashed symbol lines, in order to denote the difference to owned elements.

7.3.2 Metamodel Interfacing Language

In this section, we cover the interface definition part of the metamodel modularisation language. Clearly, a fragment that doesn't have explicit interfaces may not be regarded as a black-box fragment. Therefore, the interface definition language introduces constructs for the interfacing of concrete reusable metaclasses, in order to support the information hiding, i.e. the

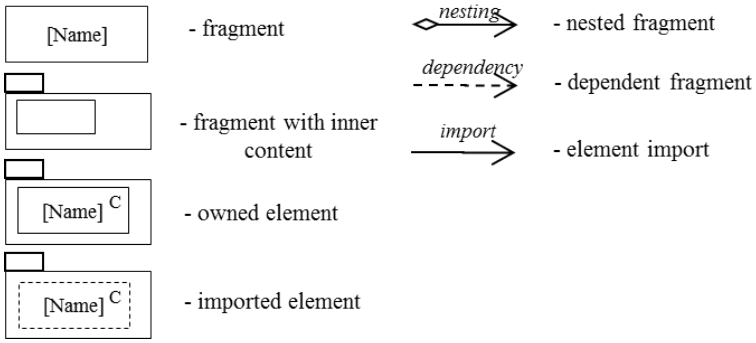


Figure 7.6: A concrete syntax of the metamodel encapsulation language (simplified specification)

idea of black-box fragments. In the following, we elaborate on the syntax, the semantics and the notation of the interfacing language.

Syntax

As we now it from programming languages, an interface exists only for classes and its main purpose is to expose or to define properties and methods an implementing class should support. Applied on metamodelling languages, the concept of interface is applicable for all main constructs, since all of them have structural features. In the core meta-metamodel, all metaclasses have attributes as their structural features. Moreover, another compound metaclass such as a model type contains additional structural features such as classes and relations, for which it would also be desirable that they are exposable via interfaces. Thus, all concrete elements that are reusable within the fragment may also be accessible for inter-fragment reuse via explicit interfaces. For example, it is desirable to expose an interface of a model type UseCaseDiagram, which other fragments may use as a connection point to support the modelling of Use Cases. In the following, the syntax schema and additional constraints are discussed.

Syntax Schema The requirement that each concrete metamodel element should be exposable via interface influences the syntax of the modularisation extension considerably. Basically, for each concrete metaclass, an appropriate interface metaclass must exist. Hence, the interfacing extension basically mimics the syntax structure of the core part to allow for interfacing of elements. Thus, we extend the abstract core metamodelling language with appropriate interface types. Basically, each abstract metamodelling concept is specialised by an adequate interface type counterpart. Figure 7.7 illustrates the extension of the core meta-metamodel that addresses the interface part of the modularisation.

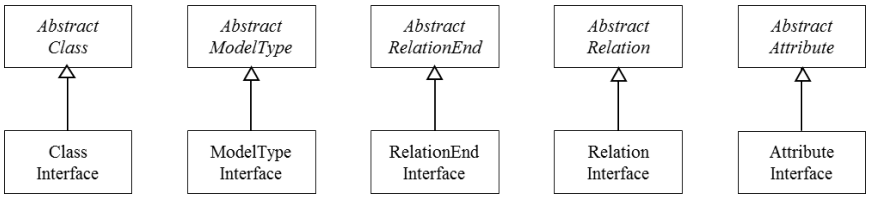


Figure 7.7: Metamodel of the metamodel modularisation language - Interface module

- *AttributeInterface.* An attribute interface is an interface of an attribute. Unlike a member attribute in GPLs, which doesn't have an interface counterpart, here, an attribute has an explicit interface. In particular, an attribute interface as a contract specifies the data type the implementing attribute has to support. Furthermore, by virtue of having an attribute interface as an explicit metamodel element, we can avoid name-based matching of member attributes of other compound interface realisations. We define the metaclass *AttributeInterface* as a subclass of the corresponding metaclass *AbstractAttribute*.

- *ClassInterface.* A class interface specifies exposable structural features of a class. In particular, a class interface specifies which attribute interfaces a concrete class has to support. To support this, we define the metaclass *ClassInterface* as a subclass of the corresponding metaclass *AbstractClass*.

- *ModelTypeInterface.* A model type interface specifies which attribute interfaces, as well as which class and relation interfaces, a concrete model type must implement in order to realise that interface. Hence, we define the metaclass *ModelTypeInterface* as a subclass of the metaclass *AbstractModelType*, in order to inherit all of the structural feature definitions.

- *RelationInterface.* A relation interface specifies which attribute interfaces and relation end interfaces must be supported by a concrete relation metamodel element, in order to be a compatible implementation of that interface. Accordingly, we define the metaclass *RelationInterface* as a subclass of the corresponding metaclass *AbstractRelation*.

- *RelationEndInterface.* A relation end interface prescribes attribute interfaces and target interfaces that must be implemented by a concrete relation end when realising that interface. Finaly, we define the metaclass *RelationEndInterface* as a subclass of the corresponding metaclass *AbstractRelationEnd*.

Constraints In Table 7.3 a list of constraints for the syntax schema of the metamodel interface definition language is provided which adds to the well-formedness of modular metamodel definitions. In particular, we define some constraints that further restrict the abstract core metamodelling language. Those constraints address mainly the relationships defined in the abstract metamodelling language which arise from the fact that both concrete and interface metaclasses are subclasses of abstract construct metaclasses leading to some not allowed combinations.

Table 7.3: Abstract syntax constraints of the interfaces part of the modularisation language

#	Constraint	Description
C14	*Attribute interface aggregation*	Any attributable interface element such as class interface and alike can only aggregate attribute interfaces as members using the *attributes* relationship.
C15	*Class interface aggregation*	A model type interface element can only aggregate class interfaces as members using the *classes* relationship.
C16	*Relation interface aggregation*	A model type interface element can only aggregate relation interfaces as members using the *relations* relationship.
C17	*Relation end interface aggregation*	A relation interface element can only aggregate relation end interfaces as members using the *relationEnds* relationship.
C18	*Relation end target interface aggregation*	A relation end interface can only aggregate target interfaces as members using the *targets* relationship.
C19	*Attribute inheritance*	The *inheritance* is not allowed relationship between interface elements.

Semantics

The concrete language semantics is, where not constrained, inherited from the abstract core metamodelling language.

Static Semantics No additional static semantic rules are defined. The static semantics of the abstract language apply.

Dynamic Semantics No additional dynamic semantic rules are defined. The dynamic semantics of the abstract language apply.

Notation

The concrete syntax of the interface part of the metamodel modularisation language may be based on the subset of the graphical syntax of the UML2 class diagrams as defined in [OMG, 2011b] as depicted in Figure 7.8. The interface circle symbol is extended with stereotypes to differentiate between different interface-like metaclasses. Alternatively, class symbols may be used for interfaces with an appropriate stereotype denoting the interface type. In addition, the "ball and socket" notation of UML2 may be used to visualise provided and required interfaces, respectively. Note, we adopt the notion of a port (small white-box) of UML composite structures to be part of required and provided interface when used to depict fragment-level interfaces. As port represents a kind of a connection point, it is convenient to connect inner elements of a fragment to the port of the interface to visualise the interface usage or realisation. Obviously, only one interface per port is allowed.

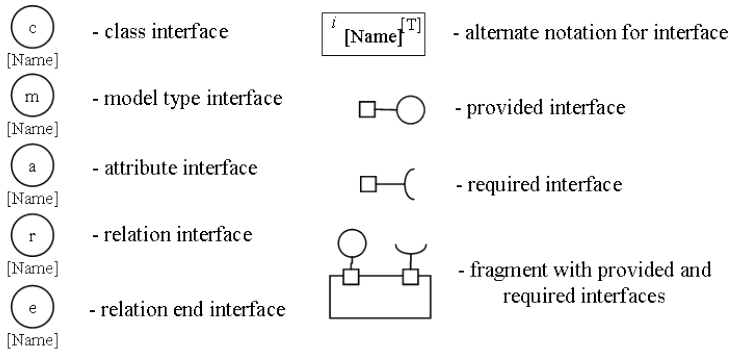


Figure 7.8: A concrete syntax of the metamodel interfacing language including the component-based notation of fragments with provided and required interfaces (simplified specification)

7.4 Metamodel Composition Language (MCL)

In this section the metamodel composition language is introduced. It is an extension of the core metamodelling language related to compositional aspects of modular metamodel engineering. The language is dependent on both the core metamodelling language and the modularisation language. Referring to the requirements on metamodel composition postulated in Chapter 6, the goal of the composition language is to provide a set of composition operators to support different types of metamodel elements (RC1) and different composition types (RC2) such as the black-box composition, the

grey-box composition and the white-box composition. Also, not only non-invasive composition should be supported, but also the invasive composition, in particular, when composing grey-boxes and white-boxes (RC3). We divide the language accordingly into sublanguages covering different types of metamodel composition approaches.

7.4.1 Black-Box Metamodel Composition Language

Composition of metamodels as black-boxes requires the existence of well-defined metamodel fragments with explicit interfaces. Hence, the language for black-box composition is dependent on both the metamodel encapsulation and the metamodel interfacing language. In the following, we introduce this language by discussing its syntax, semantics and notation.

Syntax

As elaborated in the previous chapter, there are two composition operators for black-box composition - *interface realisation* and *interface subtyping*. To recall, the *interface realisation* is a composition operation that binds an interface to its implementation. As defined before, an interface prescribes the members a concrete element must implement in order to realise that interface. Hence, the interface realisation has the semantics of ensuring the fulfillment of that interface contract. Considering the compositional structure of elements and interfaces, a concrete element realises an interface only if it realises all member interfaces. Furthermore, a concrete element may realise an arbitrary number of interfaces. This an important property of interface realisation as it allows for *interface segregation*, as one of the basic principles of interface design. In turn, an interface may be realised by an arbitrary number of implementation elements. This kind of multiple interface implementation contributes to more flexibility in metamodel composition. On the other side, *interface subtyping* relation with its substitutional and extensional semantics contributes further to the flexibility of interface realisation. An element that realises an interface, transitively represents the realisation of all of the interface supertypes. In the following, we discuss these two composition operators in terms of syntax and constraints.

Syntax Schema For each of the black-box composition operators an appropriate syntactic relation function with dedicated semantics is introduced. Since different concrete element and interface types have specific semantics with respect to composition, we define composition operators for interface realisation and interface subtyping as relationships for each core construct (class, attribute, model type, relation, relation end)[3]. The syntax schema

[3]One could have generalised these operators and defined them on the top of the element hierarchy, i.e. on the abstract metamodel element construct, however such generalisation

imports metaclasses from the concrete metamodelling language and from the interfacing language that represent implementation elements and their interface counterparts, respectively. As mentioned, for each concrete element-interface pair an appropriate interface realisation relationship is defined. In addition, for each metaclass representing an interface, an adequate, reflexive interface subtyping relationship is added. Figure 7.9 illustrates the syntax schema.

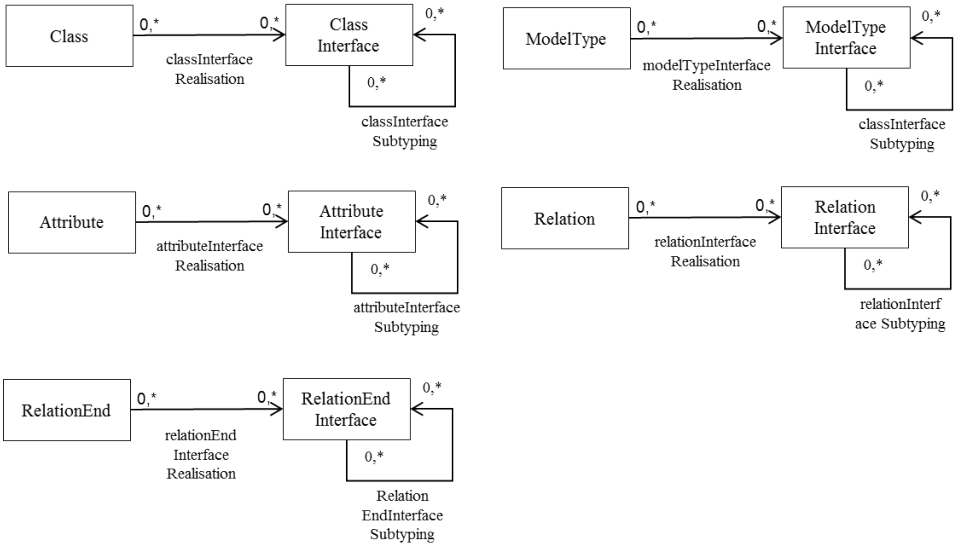


Figure 7.9: Metamodel of the black-box composition operators

We define the interface realisation relationships as follows:

- *attributeInterfaceRealisation.* An attribute interface realisation connects a concrete attribute with an attribute interface to denote the attribute interface realisation. The implementing attribute must have the attribute type that is compatible with that defined in the interface it realises. While equal type matching is trivial, compatibility of types must also hold such that the implementation type can never be more restrictive than the interface type. For example, an attribute A_1 of type integer is compatible to the attribute interface AI_1 of type unsigned integer. In addition, a concrete attribute must adhere to any additional properties of the attribute element defined in the interface[4]. Hence, we define the relationship *attributeInterfaceRealisation*

would lead to an extensive set of constraints and complex semantics, in order to capture the particularity of each core metaclass.

[4]For example, in the ADOxx Meta[2]-Model, the attribute concept has the property *language invariant* which specifies whether the value of that attribute should support multi-lingual values.

as a directed relationship from the metaclass *Attribute* to the metaclass *AttributeInterface*.

- *classInterfaceRealisation*. A class interface realisation connects a concrete class element with a class interface to express the implementation of the class interface. In order to support the interface, the implementing class must implement all attribute interfaces prescribed by that interface. For example, a class C_1 with an attribute A_1 realises a class interface CI_1, only if its attribute also realises the attribute interface AI_1. Similar to the attribute interface, all properties of an abstract class element defined at the interface must also be supported by the implementing class (refer to the class property *repository class* mentioned earlier). We define the relationship *classInterfaceRealisation* as a directed relationship from the metaclass *Class* to the metaclass *ClassInterface*.

- *modelTypeInterfaceRealisation*. A model type interface realisation connects a concrete model type with a model type interface to express the interface implementation. This composition operation is rather complex as it subsumes the realisation of all structural features (member interfaces) of the model type interface. First, all attribute interfaces must be supported by the implementing model type. Second, all class and relation interfaces must be supported, too. Finally, the implementing model type must adhere to any properties defined at the model type interface. Accordingly, we define the relationship *modelTypeInterfaceRealisation* as a directed relationship from the metaclass *ModelType* to the metaclass *ModelTypeInterface*.

- *relationInterfaceRealisation*. A relation interface realisation declares the implementation of a relation interface by a concrete relation. Like all attributable elements, the implementing relation must support all attribute interfaces defined at the relation interface. In addition, the relation ends have to support corresponding interface counterparts as well. Likewise, property matching must hold. The relationship *relationInterfaceRealisation* as a directed relationship from the metaclass *Relation* to the metaclass *RelationInterface*.

- *relationEndInterfaceRealisation*. Finally, a relation end interface establishes the realisation relationship between a concrete relation end and its interface equivalent. The realisation refers to the attribute interfaces and the target interfaces (classes or model types) prescribed at the relation end interface. Likewise, property matching must hold. Finally, the relationship *relationEndInterfaceRealisation* is a directed relationship from the metaclass *RelationEnd* to the metaclass *RelationEndInterface*.

We define the interface subtyping relationships as follows:

- *attributeInterfaceSubtyping.* An attribute interface subtyping expresses the compatibility, i.e. the substitutability of a supertype by its subtype interface. Clearly, the compatibility of the associated attribute types must hold. Since attribute doesn't have any additional member elements, there is no inheritance semantics that apply. The relationship *attributeInterfaceSubtyping* is a directed, reflexive relationship from and to the metaclass *AttributeInterface*.

- *classInterfaceSubtyping.* A class interface subtyping combines two class interfaces into subtype-supertype relationship with known substitutional semantics. In addition, a subtype interface inherits all attribute interfaces as structural features and may add additional ones. The relationship *classInterfaceSubtyping* is a directed, reflexive relationship from and to the metaclass *ClassInterface*.

- *modelTypeInterfaceSubtyping.* A model type interface subtyping declares one interface to be the subtype of the other model type interface. In doing so, it inherits all attribute interfaces, but also all class and relation interfaces. The relationship *modelTypeInterfaceSubtyping* is a directed, reflexive relationship from and to the metaclass *ModelTypeInterface*.

- *relationInterfaceSubtyping.* By subtyping the relation interface, a subtype interface inherits all attribute interfaces and relation end interfaces of the supertype. The relationship *relationInterfaceSubtyping* is a directed, reflexive relationship from and to the metaclass *RelationInterface*.

- *relationEndInterfaceSubtyping.* A relation end interface subtyping declares the subtype interface to be compatible to the supertype interface by inheriting all of its attribute interfaces, as well as target interfaces (class or model type interfaces). The relationship *relationEndInterfaceSubtyping* is a directed, reflexive relationship from and to the metaclass *RelationEndInterface*.

Constraints Table 7.4 lists the abstract syntax constraints that further restrict the syntax schema.

Semantics

The semantics of the interface realisation and interface subtyping composition operators complement the definitions provided in the abstract syntax.

Static semantics

Table 7.4: Abstract syntax constraints of the black-box composition part of the composition language

#	Constraint	Description
C13	*Acyclic subtyping dependencies*	For all subtyping relationships, an element interface cannot be its own direct supertype nor form indirect, transitive cyclic dependencies.

Definition 20 (Attribute Interface Realisation) *Given the attribute A, its data type T_1, the attribute interface AI and its data type T_2, A realises AI only if T_1 is a compatible type to T_2.*

Definition 21 (Class Interface Realisation) *Given the class C, with the set of attributes Sa and the class interface CI with the set of attribute interfaces Sai, C realises CI only if for each of the attribute interfaces in Sai a matching realisation attribute in Sa exists.*

Definition 22 (Model Type Interface Realisation) *Given the model type MT, with a set of attributes Sa, the set of classes Sc, the set of relations Sr, the model type interface MTI with a set of attribute interfaces Sai, the set of class interfaces Sci and the set of relation interfaces Sri, MT realises MTI only if each of the attribute interfaces in Sai has a realisation attribute in Sa, for each class interface in Sci there is a realisation class in Sc and for each relation interface in Sri there is a realisation relation in Sr.*

Definition 23 (Relation Interface Realisation) *Given the relation R, with a set of attributes Sa, the set of relation ends Sre, the relation interface RI with a set of attribute interfaces Sai, the set of relation end interfaces Srei, R realises RI only if each of the attribute interfaces in Sai has a realisation attribute in Sa and for each relation end interface in Srei there is a realisation relation end in Sre.*

Definition 24 (Relation End Interface Realisation) *Given the relation end RE, with a set of attributes Sa, the set of targets St, the relation end interface REI with a set of attribute interfaces Sai, and the set of target interfaces Sti, RE realises REI only if each of the attribute interfaces in Sai has a realisation attribute in Sa and a for each target interface in Sti there is a realisation target in St.*

Definition 25 (Interface Subtyping Transitiveness) *Interface subtyping is transitive relation (applicable for all interface subtyping relations). Given the interfaces I_1, I_2 and I_3, if an interface I_1 is subtype of I_2 and I_2 is subtype of I_3, then I_1 is subtype of I_3.*

Definition 26 (Interface Subtyping Reflexiveness) *Interface subtyping is reflexive relation. Given the interface I, I is subtype of I.*

Definition 27 (Attribute Interface Subtyping) *Given the attribute interface AI_1, its data type T_1, the attribute interface AI_2 and its data type T_2, AI_1 is a valid subtype of AI_2 only if T_1 is a compatible type to T_2.*

Definition 28 (Class Interface Subtyping) *Given the class interfaces CI_1 and CI_2 with corresponding sets of attribute interfaces Sai_1 and Sai_2, CI_1 is a subtype of CI_2 only if for each of the attribute interfaces in Sai_2 the same or valid subtype attribute interface in Sa_1 exists.*

Definition 29 (Model Type Interface Subtyping) *Given the model type interface MTI_1, with a set of attribute interfaces Sai_1, the set of class interfaces Sci_1, the set of relation interfaces Sri_1, the model type interface MTI_2 with a set of attribute interfaces Sai_2, the set of class interfaces Sci_2 and the set of relation interfaces Sri_2, MTI_1 is a subtype of MTI_2 only if for each of the attribute interfaces in Sai_2 the same or valid subtype attribute interface in Sa_1 exists, for each class interface in Sci_2 the same or valid subtype class interface in Sci_1 exists and for each relation interface in Sri_2 the same or valid subtype relation interface in Sri_1 exists.*

Definition 30 (Relation Interface Subtyping) *Given the relation interface RI_1, with a set of attribute interfaces Sai_1, the set of relation end interfaces $Srei_1$, the relation interface RI_2 with a set of attribute interfaces Sai_2, the set of relation end interfaces $Srei_2$, RI_1 is a subtype of RI_2 only if for each of the attribute interfaces in Sai_2 the same or valid subtype attribute interface in Sa_1 exists and for each relation end interface in $Srei_2$ the same or valid subtype relation end interface in $Srei_1$ exists.*

Definition 31 (Relation End Interface Subtyping) *Given the relation end interface REI_1, with a set of attribute interfaces Sai_1, the set of target interfaces Sti_1, the relation end interface REI_2 with a set of attribute interfaces Sai_2, and the set of target interfaces Sti_2, REI_1 is a subtype of REI_2 only if for each of the attribute interfaces in Sai_2 the same or valid subtype attribute interface in Sa_1 exists and for each target interface in Sti_2 the same or valid subtype target interface in Sti_1 exists.*

Definition 32 (Derived Relation End Interface Targets) *If a class interface or a model type interface is a target of some relation end interface, then all its subtypes are allowed target interfaces for that relation end interface.*

Definition 33 (Derived Relation End Targets) *If a class interface or a model type interface is a target of some concrete relation end, then all its concrete realisations are valid targets of that relation end.*

Dynamic semantics

Definition 34 (Derivation of Attribute Interfaces) *Given the attributable element interfaces AEI_1 and AEI_2 with corresponding sets of attribute interfaces Sai_1, Sai_2, and the interface subtyping relation R_s such that AEI_1 is subtype of AEI_2, then AEI_1 aggregates all attribute interfaces from Sai_2 into Sai_1.*

Definition 35 (Derivation of Class and Relation Interfaces) *Given the model type interfaces MTI_1 and MTI_2 with corresponding sets of class interfaces Sci_1, Sci_2 and relation interfaces Sri_1, Sri_2, and the interface subtyping relation R_s such that MTI_1 is subtype of MTI_2, then MTI_1 aggregates all class interfaces from Sci_2 into Sci_1, and all relation interfaces from Sri_2 into Sri_1.*

Definition 36 (Derivation of Relation End Interfaces) *Given the relation interfaces RI_1 and RI_2 with corresponding sets of relation end interfaces $Srei_1$, $Srei_2$, and the interface subtyping relation R_s such that RI_1 is subtype of RI_2, then RI_1 aggregates all relation end interfaces from $Srei_2$ into $Srei_1$.*

Definition 37 (Derivation of Model Type and Class Interfaces) *Given the relation end interfaces REI_1 and REI_2 with corresponding sets of model type interfaces $Smti_1$, $Smti_2$, and class interfaces Sci_1, Sci_2, and the interface subtyping relation R_s such that REI_1 is subtype of REI_2, then REI_1 aggregates all model type interfaces from $Smti_2$ into $Smti_1$, and all class interfaces Sci_2 into Sci_1.*

Notation

A possible notation of the black-box metamodel composition language may be based on the UML2 class diagram graphical syntax. In particular the UML symbols for the interface realisation and for the generalisation relationship may be used to depict the two composition operators as illustrated

in Figure 7.10. The notational differentiation of various special types of the composition operators is not needed as this is always given by the target types of those relationships. Furthermore, when a fragment is visualised as a component with exposed required and provided interfaces, the relationships to interfaces, such as interface realisation, are drawn from an inner element to the port of an interface.

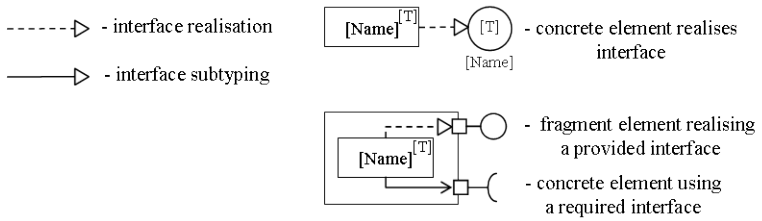


Figure 7.10: A concrete syntax of the black-box composition language including the component-based notation (simplified specification)

7.4.2 Grey-Box Metamodel Composition Language

The grey-box metamodel composition composes metamodels based on implicit interfaces. It introduces a new composition operator that is able to compose metamodel elements by injecting extensions into implicitly available extension points. The grey-box metamodel composition language depends on the extended concrete core metamodelling language. In the following, we discuss the syntax, the semantics and the notation of this language. For the easier definition of operators, we assume that metamodel elements in fragments by default are all public.

Syntax

To recall, the extension composition operator takes as input a base element and an extender (composer) element and injects structural features of the extender into the base element. The base element can be any existing compound element (that has structural features). The operator relies on the notion of dynamic invasive composition.

Syntax Schema Since each of the core metaclasses contains implicit extension points unique to that particular metaclass, it would be cumbersome to define a generic extender element which has knowledge about all possible implicit extension points of all core metaclasses. One solution would be to introduce a dedicated extender metaclass by inheriting from each of the core metaclasses, as this way, we inherit the intrinsic knowledge about their extension points. However, since the only semantic difference of the extender metaclass is that it must be an abstract, non-instantiable element, we

decide to reuse the core metaclasses to represent extender elements, themselves. Thus, to enable extension-based composition, we introduce five explicit extension composition operators as reflexive relationships for each of the core metaclasses such as *Class*, *Attribute*, *ModelType*, *Relation* and *RelationEnd*. We define that an extender element (must be abstract) may extend many base elements of the same metaclass. Contrary, a base element may be extended by many extender elements (of the same metaclass). Unlike the inheritance, the semantics of the extension relationship is that it adds member elements from an extender directly to the base element by virtue of aggregation (inverse inheritance). Figure 7.11 illustrates the syntax schema. Note that this language extension is invasive for the concrete core metamodelling language as we add new relations to each of the core metaclasses.

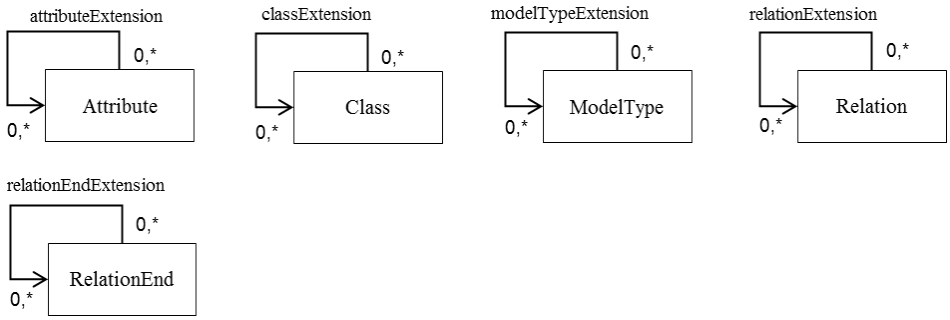


Figure 7.11: Metamodel of the grey-box composition operators

Basically, implicit extension points of core metaclasses are structural feature sets, as they represent parts of element definitions that have an extensive character. For example, the metaclass *Class* has as an extension point the set of contained attributes, which an extender element may extend. In the following, we define the implicit extension points of each of the core metamodelling constructs:

- *Class extension points.* A class is a container of attributes and supported class interfaces. Thus, the extension points are: attribute set, class interface set.
- *Model type extension points.* A model type is a container of attributes, classes, relations and model type interfaces. The implicit interfaces are: attribute set, class set, relation set, model type interface set.
- *Relation extension points.* A relation is a container of attributes, relation ends and relation interfaces, with corresponding extension points being: attribute set, relation end set, relation interface set.

- *Relation end extension points.* Like other attributable and interfacable elements relation end allows for attribute extensions and relation end interface extensions. In addition, the relation end has the allowed target set as unique implicit interface for this type.

The corresponding five extension composition operators are defined as follows:

- *attributeExtension.* An attribute extension extends a base attribute by an attribute extender at the attribute interfaces extension point. We define the relationship *attributeExtension* as a directed, reflexive relationship from and to the metaclass *Attribute*.

- *classExtension.* A class extension extends a base class element by a class extender by adding new attributes and supported class interfaces. The relationship *classExtension* is a directed, reflexive relationship from and to the metaclass *Class*.

- *modelTypeExtension.* A model type extension operator extends a base model type element by its extender counter part by injecting allowed structural features such as attributes, classes, relations and model type interfaces. The relationship *modelTypeExtension* is a directed reflexive, relationship from and to the metaclass *ModelType*.

- *relationExtension.* A relation extension may inject attributes, relation ends and relation interfaces from an extender relation to the base relation. Accordingly, the relationship *relationExtension* is a directed, reflexive relationship from and to the metaclass *ModelType*.

- *relationEndExtension.* A relation end extension operator extends relation ends for attributes, targets and relation end interfaces. Likewise, the relationship *relationEndExtension* is a directed reflexive, relationship from and to the metaclass *RelationEnd*.

Constraints Table 7.5 lists the abstract syntax constraints that further restrict the syntax schema of the grey-box composition constructs. The constraints arise from the reflexive nature of the extension relation and the fact that the core element types play the role of a base element and an extender. In addition, the combined use of inheritance and extension implies certain restrictions.

Semantics

The semantics of the grey-box composition operators is mainly of the dynamic nature. It defines how the elements defined at the extender part are injected to the corresponding base elements based on the inherent extension points of metaclasses.

Table 7.5: Abstract syntax constraints of the grey-box composition part of the composition language

#	Constraint	Description
C60	*Acyclic extension dependency*	For all extension relationships applies, an extender cannot extends itself, neither directly nor indirectly.
C61	*Abstract extender element*	For all metaclasses for which extension relationship exists, a metamodel element can be an extender element only if it is declared as abstract.
C62	*Acyclic extension-inheritance dependency*	For all metaclasses for which extension and inheritance relationships exist, an extender metamodel element cannot extend a base element, if it at the same time inherits from it, neither directly nor indirectly.

Static semantics Besides the intrinsic semantics of the core elements, we define the semantics of the extension relationships.

Definition 38 (Attribute Extension) *Given the attributes A_1 and A_2 with corresponding sets of attribute interfaces Sai_1, Sai_2, A_1 extends A_2 only if for each of the attribute interfaces in Sai_1 the same attribute interface in Sai_2 exists.*

Definition 39 (Class Extension) *Given the classes C_1 and C_2 with corresponding sets of attributes Sa_1, Sa_2, and with corresponding sets of class interfaces Sci_1, Sci_2, C_1 extends C_2 only if for each of the attributes in Sa_1 the same attribute in Sa_2 exists, and for each of the class interfaces in Sci_1 the same class interface in Sci_2 exists.*

Definition 40 (Model Type Extension) *Given the model types MT_1 and MT_2 with corresponding sets of attributes Sa_1, Sa_2, with corresponding sets of classes Sc_1, Sc_2, with corresponding sets of relations Sr_1, Sr_2, and with corresponding sets of model type interfaces $Smti_1$, $Smti_2$, MT_1 extends MT_2 only if for each of the attributes in Sa_1 the same attribute in Sa_2 exists, for each of the classes in Sc_1 the same class in Sc_2 exists, for each of the relations in Sr_1 the same relation in Sr_2 exists, and for each of the model type interfaces in $Smti_1$ the same model type interface in $Smti_2$ exists.*

Definition 41 (Relation Extension) *Given the relations R_1 and R_2 with corresponding sets of attributes Sa_1, Sa_2, with corresponding sets of relation ends Sre_1, Sre_2, and with corresponding sets of relation interfaces Sri_1, Sri_2, R_1 extends R_2 only if for each of the attributes in Sa_1 the same attribute in Sa_2 exists, for each of the relation ends in Sre_1 the same relation end in Sre_2 exists, and for each of the relation interfaces in Sri_1 the same relation interface in Sri_2 exists.*

Definition 42 (Relation End Extension) *Given the relation ends RE_1 and RE_2 with corresponding sets of attributes Sa_1, Sa_2, with corresponding sets of targets St_1, St_2, and with corresponding sets of relation end interfaces $Srei_1$, $Srei_2$, RE_1 extends RE_2 only if for each of the attributes in Sa_1 the same attribute in Sa_2 exists, for each of the targets in St_1 the same target in St_2 exists, and for each of the relation end interfaces in $Srei_1$ the same relation end interface in $Srei_2$ exists.*

Definition 43 (Transitiveness of Extension) *For all extension relations applies, given the metamodel elements E_1, E_2 and E_3, if an Element E_1 extends E_2 and E_2 extends E_3, then E_1 extends E_3.*

Definition 44 (Reflexiveness of Extension) *Extension is reflexive relation. For any metamodel element E_1, E_1 extends E_1.*

Definition 45 (Derived Relation End Targets) *By virtue of extension, if an extender class or model type is a target of some relation end, the extended elements of the extender elements are valid targets of that relation end. Given the relation end RE_1 and the set of targets St_1, the extender element (class or model type) E_1, and the element E_2, such that E_1 is a valid target of RE_1 (exists in St_1), if E_1 extends E_2, then E_2 is also a valid target of RE_1.*

Dynamic semantics

Definition 46 (Injection of Attributes) *Given the attributable elements AE_1 and AE_2 with corresponding sets of attributes Sa_1, Sa_2, and the extension relation R_e such that AE_1 extends AE_2, then AE_2 aggregates all attributes from Sa_1 into Sa_2.*

Definition 47 (Injection of Classes and Relations) *Given the model types MT_1 and MT_2 with corresponding sets of classes Sc_1, Sc_2 and relations Sr_1, Sr_2, and the extension relation R_e such that MT_1 extends MT_2, then MTI_2 aggregates all classes from Sc_1 into Sci_2, and all relations from Sr_1 into Sr_2.*

Definition 48 (Injection of Relation Ends) *Given the relations R_1 and R_2 with corresponding sets of relation ends Sre_1, Sre_2, and the extension relation R_e such that R_1 extends R_2, then R_2 aggregates all relation ends from Sre_1 into Sre_2.*

Definition 49 (Injection of Model Types and Classes) *Given the relation ends RE_1 and RE_2 with corresponding sets of model types Smt_1, Smt_2, and classes Sc_1, Sc_2, and the extension relation R_e such that RE_1 extends RE_2, then RE_2 aggregates all model types from Smt_1 into Smt_2, and all classes from Sc_1 into Sc_2.*

Definition 50 (Injection of Interfaces) *For every interfaceable concrete element type and its corresponding interface type applies, given the elements E_1 and E_2 with corresponding sets of interfaces Si_1, Si_2, and the extension relation R_e such that E_1 extends E_2, then E_2 aggregates all interfaces from Si_1 into Si_2.*

Notation

A possible notation of the grey-box metamodel composition language may be based on the UML class diagram graphical notation. The extension operators are directed association symbols annotated with an appropriate stereotype *"extends"*. No additional differentiation for different types of extensions is necessary (classExtension, attributeExtension etc.), as this is implicitly derivable from the connecting elements. Figure 7.13 illustrates the extension relationship notation.

7.4.3 White-Box Metamodel Composition Language

The white-box metamodel composition operates on the exposed concrete metamodel elements of metamodel fragments. Standard white-box composition operations that are already part of the core metamodelling language are inheritance and aggregation. To complement these operators, we introduce the *mixin inclusion* composition operator which is the main contribution of our white-box metamodel composition language. In the following, we discuss the syntax, the semantics and the notation of such white-box metamodel composition language. For the easier definition of the new operator, we assume that metamodel elements in fragments by default are all public.

Syntax

Syntax Schema The mixin inclusion composition operator combines white-box concrete metamodel elements. Hence, metaclasses that represent core concrete elements from the core language are taken as elements on which

the composition takes place. The operator is defined reflexively on the level of a metamodel element. By doing so, this extension module itself invasively modifies each of the concrete elements of the core metamodelling language by adding the corresponding mixin inclusion operator relationships. Figure 7.12 illustrates the syntax schema of the white-box composition language.

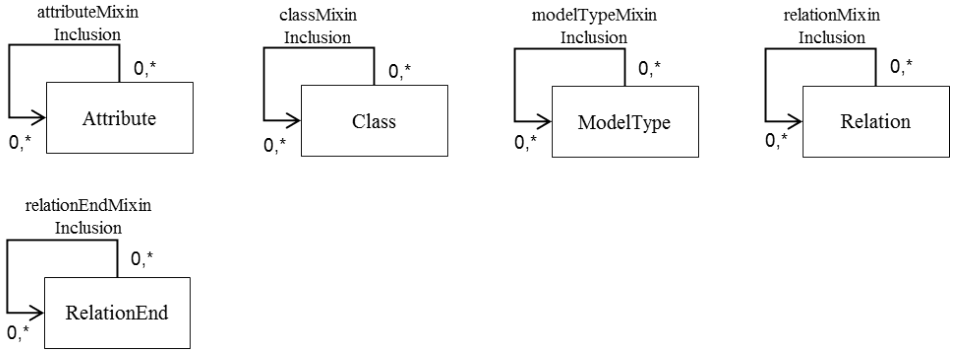


Figure 7.12: Metamodel of the white-box composition operators

Basically, the mixin operator includes (or "mixes in") structural features of a *mixin element* into a *base element*. A mixin element must be declared as non-instantiable abstract metamodel element. A base element may mix in many mixins. Likewise, a mixin element may be mixed in by different base elements. As for including the elements, the mixin inclusion operator relies on the aggregation composition operator applied on the set of elements. Which structural features are mixed in is defined by the corresponding concrete composition operator.

The mixin inclusion relationships are defined as follows:

- *attributeMixinInclusion*. An attribute mixin inclusion includes attribute interfaces of a mixin attribute into a base attribute. We define the relationship *attributeMixinInclusion* as a directed, reflexive relationship from and to the metaclass *Attribute*.

- *classMixinInclusion*. A class mixin inclusion mixes in structural features of a class such as attribute and class interface of a mixin class into a base class. The relationship *classMixinInclusion* is a directed, reflexive relationship from and to the metaclass *Class*.

- *modelTypeMixinInclusion*. A model type mixin inclusion allows a base model type to mix in attributes, classes, relations and model type interfaces of a mixin model type. The relationship *modelTypeMixinInclusion* is a directed, reflexive relationship from and to the metaclass *ModelType*.

- *relationMixinInclusion.* A relation mixin inclusion mixes in attributes, relation ends and relation interfaces of a mixin relation into a base relation. The relationship *relationMixinInclusion* is a directed, reflexive relationship from and to the metaclass *Relation.*

- *relationEndMixinInclusion.* A relation end mixin inclusion allows a base relation end to include attributes, connectable targets and relation end interfaces of a mixin relation end. Likewise, the relationship *relationEndMixinInclusion* is a directed, reflexive relationship from and to the metaclass *RelationEnd.*

Since we define the mixin inclusion operator for each metaclass, the same type restriction is inherently given. For example, one cannot use the relationship *classMixinInclusion* from a model type to mix in classes. However, in some meta-languages, mixin inclusion may require that some additional metaclass properties have to match between the mixin element and the parent element[5]. However we do not specify which specific metaclass properties must be compatible as the list of properties for each core element may vary between different metamodelling languages.

Constraints Table 7.6 lists the abstract syntax constraints that further restrict the syntax schema of the white-box composition constructs.

Semantics

The semantics of the white-box composition operator extends the semantics of the concrete metamodelling language and is defined as follows.

Static semantics The static semantics as defined in *Definitions 51-58* introduces basic semantic characteristics of the mixin inclusion operator.

Definition 51 (Attribute Mixin Inclusion) *Given the attributes A_1 and A_2 with corresponding sets of attribute interfaces Sai_1, Sai_2, A_1 mixes in A_2 only if for each of the attribute interfaces in Sai_2 the same attribute interface in Sai_1 exists.*

Definition 52 (Class Mixin Inclusion) *Given the classes C_1 and C_2 with corresponding sets of attributes Sa_1, Sa_2, and with corresponding sets of class interfaces Sci_1, Sci_2, C_1 mixes in C_2 only if for each of the attributes in Sa_2 the same attribute in Sa_1 exists, and for each of the class interfaces in Sci_2 the same class interface in Sci_1 exists.*

[5]For example, ADOxx introduces the property "repository class" of the metaclass *Class*, which further specifies that type. See Chapter 8 for further details.

Table 7.6: Abstract syntax constraints of the white-box composition part of the composition language

#	Constraint	Description
C80	*Acyclic mixin inclusion dependency*	For all mixin inclusion relationships applies, a base element cannot mix in itself, neither directly nor indirectly.
C81	*Abstract mixin element*	For all mixin metaclasses, a metamodel element can be a mixin only if it is declared as abstract.
C82	*Acyclic inclusion-inheritance dependency*	For all mixin metaclasses for which mixin inclusion and inheritance relationships exist, a base element cannot mix in a mixin element, if a mixin element inherits from a base element, neither directly nor indirectly.
C83	*Acyclic inclusion-extension dependency*	For all mixin metaclasses for which mixin inclusion and extension relationships exist, a base element cannot mix in a mixin element, if it at the same time extends it, neither directly nor indirectly.

Definition 53 (Model Type Mixin Inclusion) *Given the model types MT_1 and MT_2 with corresponding sets of attributes Sa_1, Sa_2, with corresponding sets of classes Sc_1, Sc_2, with corresponding sets of relations Sr_1, Sr_2, and with corresponding sets of model type interfaces $Smti_1$, $Smti_2$, MT_1 mixes in MT_2 only if for each of the attributes in Sa_2 the same attribute in Sa_1 exists, for each of the classes in Sc_2 the same class in Sc_1 exists, for each of the relations in Sr_2 the same relation in Sr_1 exists, and for each of the model type interfaces in $Smti_2$ the same model type interface in $Smti_1$ exists.*

Definition 54 (Relation Mixin Inclusion) *Given the relations R_1 and R_2 with corresponding sets of attributes Sa_1, Sa_2, with corresponding sets of relation ends Sre_1, Sre_2, and with corresponding sets of relation interfaces Sri_1, Sri_2, R_1 mixes in R_2 only if for each of the attributes in Sa_2 the same attribute in Sa_1 exists, for each of the relation ends in Sre_2 the same relation end in Sre_1 exists, and for each of the relation interfaces in Sri_2 the same relation interface in Sri_1 exists.*

Definition 55 (Relation End Mixin Inclusion) *Given the relation ends RE_1 and RE_2 with corresponding sets of attributes Sa_1, Sa_2, with corresponding sets of targets St_1, St_2, and with corresponding sets of relation*

end interfaces $Srei_1$, $Srei_2$, RE_1 mixes in RE_2 only if for each of the attributes in Sa_2 the same attribute in Sa_1 exists, for each of the targets in St_2 the same target in St_1 exists, and for each of the relation end interfaces in $Srei_2$ the same relation end interface in $Srei_1$ exists.

Definition 56 (Transitiveness of Mixin Inclusion) *For all mixin inclusion relations applies, given the metamodel elements E_1, E_2 and E_3, if an Element E_1 mixes in E_2 and E_2 mixes in E_3, then E_1 mixes in E_3.*

Definition 57 (Reflexiveness of Mixin Inclusion) *Mixin inclusion is a reflexive relation. For any metamodel element E_1, E_1 mixes in E_1.*

Definition 58 (Derived Relation End Targets) *By virtue of mixin inclusion relation, if a mixin class or model type is a target of some relation end, then elements that include a mixin element are valid targets of that relation end. Given the relation end RE_1 and the set of targets St_1, the mixin element (class or model type) E_1, and the element E_2, such that E_1 is a valid target of RE_1 (exists in St_1), if E_2 mixes in E_1, then E_2 is also a valid target of RE_1.*

Dynamic Semantics The dynamic semantics of the mixin inclusion operator as defined in *Definitions 59-63* describe how the mixin inclusion operator is applied on metamodel elements.

Definition 59 (Inclusion of Attributes) *Given the attributable elements AE_1 and AE_2 with corresponding sets of attributes Sa_1, Sa_2, and the mixin inclusion relation R_m such that AE_1 mixes in AE_2, then AE_1 aggregates all attributes from Sa_2 into Sa_1.*

Definition 60 (Inclusion of Classes and Relations) *Given the model types MT_1 and MT_2 with corresponding sets of classes Sc_1, Sc_2 and relations Sr_1, Sr_2, and the mixin inclusion relation R_m such that MT_1 mixes in MT_2, then MTI_1 aggregates all classes from Sc_2 into Sci_1, and all relations from Sr_2 into Sr_1.*

Definition 61 (Inclusion of Relation Ends) *Given the relations R_1 and R_2 with corresponding sets of relation ends Sre_1, Sre_2, and the mixin inclusion relation R_m such that R_1 mixes in R_2, then R_1 aggregates all relation ends from Sre_2 into Sre_1.*

Definition 62 (Inclusion of Model Types and Classes) *Given the relation ends RE_1 and RE_2 with corresponding sets of model types Smt_1, Smt_2, and classes Sc_1, Sc_2, and the mixin inclusion relation R_m such that RE_1 mixes in RE_2, then RE_1 aggregates all model types from Smt_2 into Smt_1, and all classes from Sc_2 into Sc_1.*

Definition 63 (Inclusion of Interfaces) *For every interfaceable concrete element type and its corresponding interface type applies, given the elements E_1 and E_2 with corresponding sets of interfaces Si_1, Si_2, and the mixin inclusion relation R_m such that E_1 mixes in E_2, then E_1 aggregates all interfaces from Si_2 into Si_1.*

Notation

Similar to the previous notation proposals, the white-box metamodel composition language may be based on the UML2 class diagram graphical syntax. The mixin inclusion operator is a directed relationship symbol annotated with the stereotype *"includes"*. Likewise, no additional symbols for different types of mixin inclusion relationships are needed (classMixin, modelTypeMixin, etc.), as this is derivable from the type of the connecting elements. Figure 7.13 illustrates the notation of the mixin inclusion relationship.

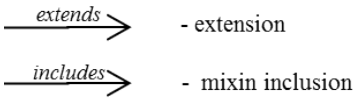


Figure 7.13: A concrete syntax of the grey-box and white-box composition operators

7.5 Chapter Summary

In this chapter, we introduced MMEL, a language for modular metamodel definition, which represents an extension to the existing metamodelling language constructs that allows for modular definition of metamodels. The MMEL itself is designed in a modular way and it consists of three basic modules for each of which a separate section has been dedicated. In Section 7.2, we introduced the core metamodelling language, which represents a common implementation of existing metamodelling constructs. The core module itself consists of an abstract and a concrete language module. This separation was important, in order to inherit certain concepts from an abstract module by other modules we introduced later. In Section 7.3 we specified the metamodel modularisation language, which itself consists of a

module for metamodel encapsulation and a module for metamodel interfacing. The encapsulation module allows for the packaging of metamodels into self-contained metamodel modules, whereas interfacing brings concepts for information hiding. Finally, in Section 7.4 the metamodel composition language has been defined featuring composition operators for interface-based black-box composition, extension-based grey-box composition and mixin-based white-box metamodel composition.

Since the architecture of MMEL is itself modular, it allows for a step-wise adoption and realisation in metamodelling platforms. For example, it is sensible to adopt the encapsulation module in the very first step, as is represents the basis for modular metamodel definition. Afterwards, modules for grey-box and white-box composition may be independently realised. In case of the black-box composition module, the interfacing module is required. Even on the level of metaclasses, the step-wise adoption may be suggested. For example, composition operators such as interface realisation, extension and mixin inclusion may initially be introduced only for class constructs followed by other metaclasses in subsequent steps. A possible partial realisation of the MMEL is elaborated in the next chapter.

Chapter 8

A Realisation of MMEL in ADOxx

"I do not think there is any thrill that can go through the human heart like that felt by the inventor as he sees some creation of the brain unfolding to success... Such emotions make a man forget food, sleep, friends, love, everything."

Nikola Tesla

A possible implementation of MMEL as introduced in the previous chapter is subject to discussion of this chapter. The ADOxx metamodelling platform (see Chapter 4) is used as the underlying metamodelling environment for the language application. In particular, the ADOxx metamodelling language (aka ADOxx-Meta^2-Model) is extended by the modularisation and composition metamodelling capabilities. Recalling the requirements for MME introduced in Chapter 6 in Section 6.1, and in particular, the requirements related to the metamodel composition technique, ADOxx is extended to natively support MME (RT1), using interpretative derivation of composite fragments (RT2) and without influencing the model-level mechanisms, i.e. the operational semantics of the existing metamodelling concepts (RT3).

The chapter is structured as follows. In Section 8.1, we introduce first the core metamodelling capabilities of ADOxx on the implementation level. Afterwards, in Section 8.2 and Section 8.3, we elaborate on how ADOxx metamodelling language is extended considering requirements RT1, RT2 and RT3, to support metamodel modularisation and composition capabilities in order to allow for modular metamodel engineering. Section 8.4 explains the applicability of the MME concepts in the context of MFBs, a broader

approach to modular modelling method engineering in ADOxx. Section 8.5 concludes the chapter.

8.1 ADOxx Metamodelling Language Implementation

We introduced the ADOxx metamodelling platform in Chapter 4 as one of the metamodel-based, interpretative metamodelling environments. ADOxx supports configuration-based tool derivation, i.e. it uses metamodels as pivotal language definition constructs to configure and instantiate the underlying model-level infrastructure. To define metamodels, ADOxx offers a metamodelling language, i.e. a meta-metamodel known as ADOxx Meta^2-Model. We discussed the metamodelling capabilities of the ADOxx Meta^2-Model on the conceptual level thoroughly in Chapter 3. In this section, we concentrate on the implementation level of the ADOxx Meta^2-Model by elaborating on how the syntax, semantics and notation of the meta-language are realised. This lays down the foundation for the extension of the language in the subsequent sections.

8.1.1 Syntax of the ADOxx Meta^2-Model

Residing on the top of the metamodelling hierarchy, the meta-metamodel of ADOxx, the ADOxx Meta^2-Model, represents the conceptual core of the ADOxx metamodelling platform, also called the library kernel. The core *syntax schema* and *syntax constraints* are implemented in C++ based on object-oriented and component-oriented design principles. All of the core constructs are exposed via the Interface Definition Language (IDL) through ADOxx APIs, that makes them available for the integration and use by other components and in other GPLs. In the course of the approach realisation, ADOxx Meta^2-Model is considered as the core metamodelling language module of the language for MME as defined in the previous chapter.

Syntax Schema

The core part of the library kernel, the syntax schema introduces basic metamodelling constructs such as library, model type, class, relation class, endpoint, attribute and attribute type[1]. Unlike the conceptual view introduced in Section 3.4, Figure 8.1 illustrates an excerpt of the implementation view of the ADOxx meta-metamodel, that reveals how single concepts are ordered hierarchically and which key member properties they feature. In ADOxx, any metamodel element construct is a subtype of the core object

[1]For the sake of simplicity, we will not consider other advanced metamodelling constructs of ADOxx such as modes, contexts, context parameters, etc.

(*IAdoCoreObject*), which guarantees that every construct has an appropriate ID for unique identification within the metamodel. Furthermore, all constructs that may have names are subtypes of the named object (*IAdoCoreNamedObject*).

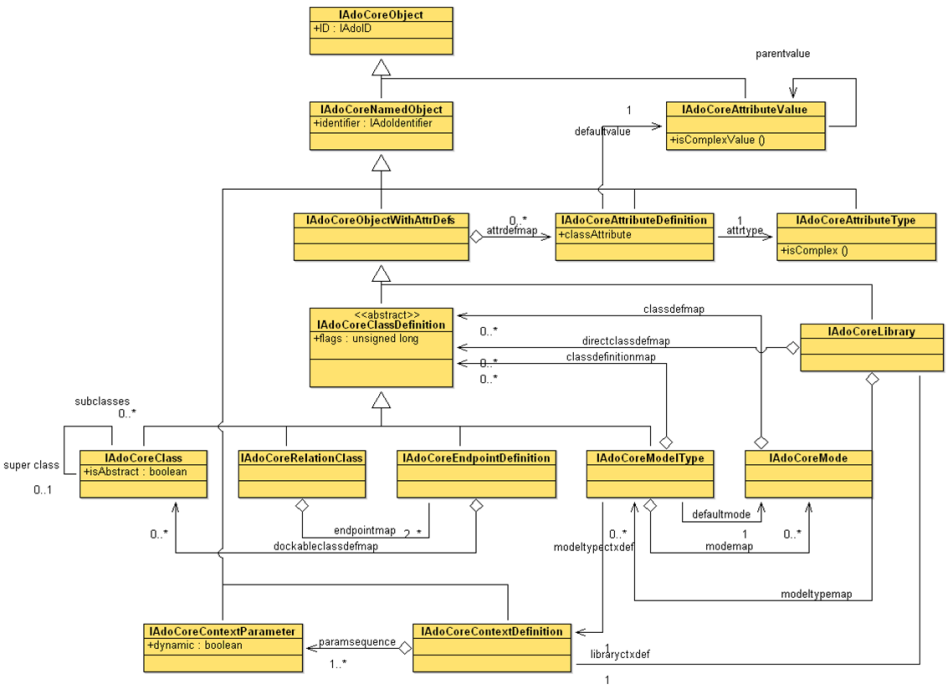


Figure 8.1: An excerpt of the implementation view of the ADOxx Meta^2-Model

While single concrete constructs have been described, here we concentrate on how ADOxx constructs are mapped to the abstract metamodelling language constructs of our approach. Table 8.1 introduces the mapping. It can be noted that all basic/core metamodelling constructs are covered, with some minor exceptions with regard to the inheritance of attributes, arity of relations and the availability of the abstract property of metaclasses. The current version of ADOxx supports inheritance of attributes for classes, binary relations and abstract property for classes.

Syntax Constraints

The syntax constraints further restrict the abstract syntax schema. In ADOxx, the syntax constraints of the ADOxx Meta^2-Model itself are part of the library kernel and are implemented as a dedicated metamodel check module (aka library checks module). Due to its component-based architecture, ADOxx allows for adding additional constraints beyond those natively supported by the metamodel kernel. An example of a typical metamodelling

Table 8.1: Mapping of abstract core metamodelling language concepts to ADOxx Meta^2-Model

Abstract Core MM-Language	**ADOxx Meta^2-Model**
Attribute	IAdoCoreAttributeDefinition
Class	IAdoCoreClass (inheritance, can be abstract)
Model type	IAdoCoreModelType
Relation	IAdoCoreRelationClass (binary)
Relation end	IAdoCoreEndpointDefinition

language syntax constraint is the *cyclic inheritance check* (See constraint *C2* in Chapter 7), which prevents creating cyclic dependencies using the inheritance relationship. Listing 8.1 shows the basic implementation of the cyclic inheritance check syntax constraint.

```
boolean hasCyclicInheritance (IAdoCoreClass aClass)
{
  IAdoCoreClass aSuper = aClass.getSuperType();
  while (aSuper)
  {
    if (aClass == aSuper)
    {
      return true;
    }
    aSuper = aSuper.getSuperType();
  }
  return false;
}
```

Listing 8.1: An example of syntax constraint of the ADOxx metamodelling language: Cyclic inheritance check

8.1.2 Semantics of the ADOxx Meta^2-Model

The semantics of the ADOxx metamodelling language is imperatively defined, i.e. it is programmed using a GPL and represents the inherent behaviour of the metamodelling tool. While some parts of the semantics of the metamodelling language specify the meaning and the behaviour of the metamodelling constructs on the metamodel-level (aka *metamodel-level semantics*), other parts of the semantics have affect on the model level (aka *model-level semantics*). For example, the metamodelling construct inheritance, if seen as a pure reuse construct in metamodelling, has its semantics defined only on the metamodel level. However, if we add the polymorphism, the behaviour of objects to be instances of several classes as long

as those classes form an inheritance hierarchy, its semantics is specified on the model level. Similarly, the behavioural meaning of an abstract class (as metamodel element) can only be defined in the model-level mechanisms (e.g. model repository, model editor, etc. see Section 4.2), which interpret such classes as being non-instantiable. In the following, we concentrate on the semantics of the metamodelling constructs that are defined only on the metamodel-level.

Static Semantics

To exemplify the implementation of the static semantics of the ADOxx metamodelling language, we refer to the transitiveness property of the attribute inheritance as defined in Section 7.2 (*Definition 2*). Basically, the transitiveness allows for a class to be not only a subclass of its direct parent class, but recursively of all parents of the parents. While the class inheritance relationship is, clearly, an intrinsic property (outgoing relationship) of a class, the transitiveness may be realised as a service/method of a *IAdoCoreClass*, which retrieves all direct and indirect parent classes of a given class. Listing 8.2 shows a simplified implementation of the inheritance transitiveness in ADOxx.

```
Set<IAdoCoreClass> getAllSuperClasses ()
{
  Set<IAdoCoreClass> aSuperSet = new HashSet<IAdoCoreClass>()
    ;
  IAdoCoreClass aSuperClass = this.getSuperClass();
  if (aSuperClass)
  {
    aSuperSet.add(aSuperClass);
    // recursively collect all direct and indirect
    superclasses
    aSuperSet.addAll(aSuperClass.getAllSuperClasses());
  }
  return aSuperSet;
}
```

Listing 8.2: An example of the implementation of the static semantics of the ADOxx metamodelling language: Transitiveness of class inheritance

Dynamic Semantics

Likewise, we refer to the inheritance of attributes as dynamic semantics example of the ADOxx metamodelling language (*Definition 5*). Basically, if a class is declared to be the subclass of another class, it inherits all of its attributes. This behaviour in ADOxx is realised by getting all attributes of the superclass and aggregating them to the subclass. Listing 8.3 shows a simplified algorithm for inheriting the attributes of the superclass.

```
void inheritAttributes (IAdoCoreClass superCLass)
{
  Iterator<IAdoCoreAttributeDefinition> aAttributes =
    superClass.getAttributes();
  while (aAttributes.hasNext())
  {
    IAdoCoreAttributeDefinition aAttr = aAttributes.next();
    this.attrDefMap.add (aAttr);
  }
}
```

Listing 8.3: An example of the implementation of the dynamic semantics of the ADOxx metamodelling language: Inheriting attributes of superclass

8.1.3 Notation of the ADOxx Meta^2-Model

As discussed in Chapter 4.2, ADOxx provides dedicated editors for language definition. In particular, the abstract syntax of a modelling language based on ADOxx Meta^2-Model may be edited using tree-based and form-based editors. Tree-based editors are used to visualise and edit the containment structures of a metamodel definition (a library contains model types, model types contain classes, etc.) and to visualise and edit the class hierarchy. Form-based editors are used to specify properties of metamodel elements. For each of the metamodelling constructs (class, model type, attribute, etc.), a form-based editor exists. Figure 8.2 shows a screenshot of tree-based editors and a form-based editor for the metamodel element class. Note that, by definition, ADOxx Meta^2-Model may be defined using the textual notation by utilising ADOxx APIs for metamodelling in some of the supported GPLs such as Java, C++ or in the scripting languages such as JavaScript.

8.2 Implementing Metamodel Modularisation in ADOxx

In Chapter 7, we introduced the metamodel modularisation language as part of the MMEL. This section elaborates on the implementation of the metamodel modularisation language as an extension of the ADOxx metamodelling language. In doing so, both sublanguages of the modularisation language are considered, the encapsulation and the interfacing language. We discussed in Chapter 3 the core and supporting capabilities of the ADOxx Meta^2-Model and came to the conclusion that ADOxx provides powerful concepts for core metamodelling. In the following, we show how the syntax, the semantics and the notation of the ADOxx Meta^2-Model may be extended to allow for metamodel modularisation regarding both encapsulation and interfacing concepts.

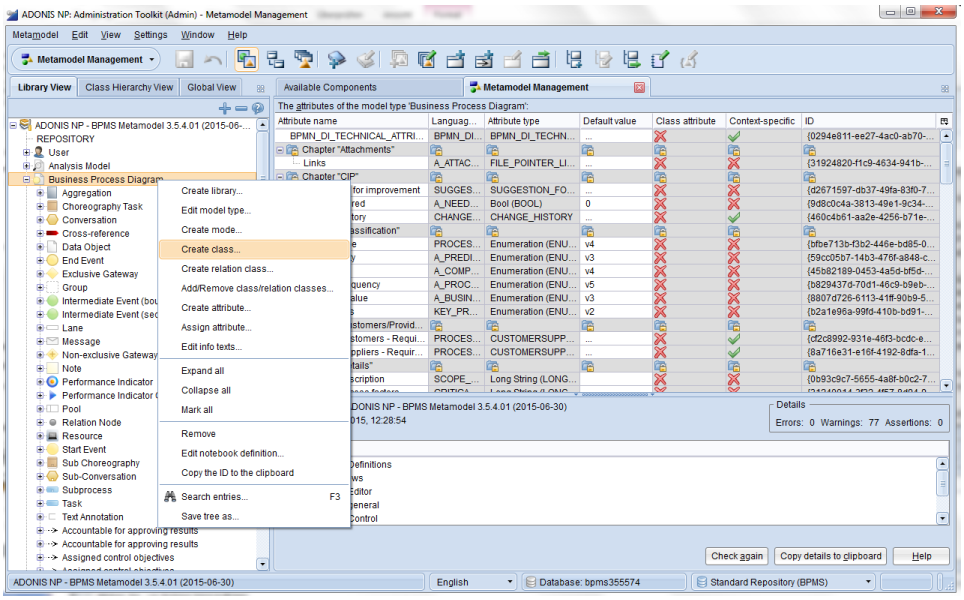


Figure 8.2: A screenshot of the tree-based and form-based notation of the ADOxx Meta^2-Model

8.2.1 Extending the Syntax

The fundamental syntactic notions of the modularisation language are the *metamodel fragment* to support encapsulation and the notion of *interface* to support interfacing. A metamodel fragment is simply a package construct that may *contain metamodel elements* and/or other fragments and may *import* and use elements from other fragments based on *explicit dependencies*. On the other side, an interface is a construct which introduces the level of indirection when composing metamodel elements to support information hiding and flexible composition. Note that, for the sake of simplicity, we explicate how the interface concept is realised in ADOxx only for the metaclasses class and attribute, both being pivotal concepts in metamodelling. Similar realisation is deducible for other metaclasses of a metamodelling language. In the following, we discuss how these concepts may be included into the existing syntax of ADOxx Meta^2-Model.

Syntax Schema

Encapsulation Like other main constructs, a fragment may uniquely be identified by an UUID and may have a name. Hence, we define fragment (*IAdoCoreFragment*) as a subtype of the named object(*IAdoCoreNamedObject*). Furthermore, a fragment may contain other metamodel elements. Since all core metamodel elements in the ADOxx syntax schema are subtypes of the named object, being their generalised construct, this element becomes

also the perfect candidate for the role of a containable metamodel element. Thus, we specify that a fragment is a named object that may contain other named objects. As specified in the modularisation language, a fragment may contain owned elements and imported elements from other fragments. Therefore, we add two new relationships from a fragment to the named object, namely, *ownedElements* and *importedElements*. Furthermore, a fragment may nest other fragments and depend on other fragments. These two relations, *nestedFragments* and *dependentFragments* are added as reflexive relationships on the fragment element directly. One may argue that the nesting and dependency may be also expressed via owned and imported element relationships, respectively, in case the target element is a fragment. While this is true, we keep having dedicated containment and import relationships on the fragment level and on the element level separated for the sake of precise semantics.

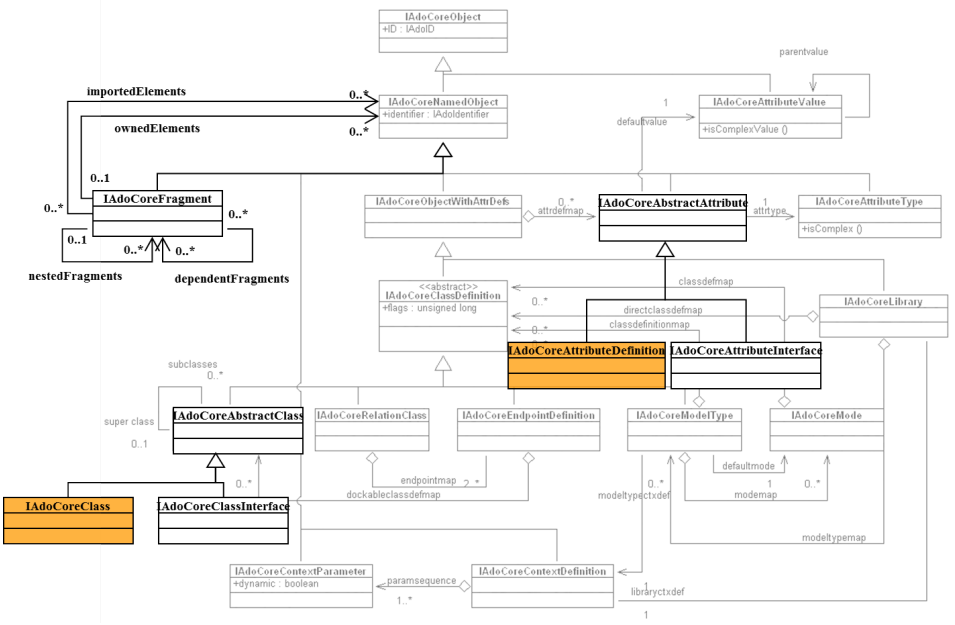


Figure 8.3: Extending the ADOxx Meta^2-Model for metamodel modularisation

Interfacing The inclusion of the interface concepts into the existing syntax structure raises two important restrictions to be considered. First, the interface concept should be injected into the existing syntax such that it can reuse the existing structural relationships of concepts (a class contains attributes, etc.). Second, extensions in the syntax must not introduce any incompatibilities with existing model-level mechanisms (changes in syntax and semantics) such as model repository. Analysing the existing syntax

schema it becomes clear that ADOxx doesn't have the abstract core concepts for each concrete core construct that would represent abstract supertypes of concrete elements and their interface counterparts as defined in Section 7.3. Therefore, we introduce two new constructs for each interfaceable element, the abstract concept and the interface concept. In particular, the abstract class *IAdoCoreAbstractClass* is an abstract non-instantiable element introduced as a superelement of the existing core class. Now, the class interface (*IAdoCoreClassInterface*) is defined as a subtype of the abstract class. Note that this change in the syntax schema doesn't introduce any semantic changes, as we simply introduced another abstract level in the class hierarchy. The same is done for the attribute concept. The abstract attribute (*IAdoCoreAbstractAttribute*) becomes the supertype of the existing concrete attribute (*IAdoCoreAttributeDefinition*) and the new attribute interface (*IAdoCoreAttributeInterface*).

Figure 8.3 shows how the modularisation constructs extend the syntax schema of the Meta^2-Model.

Syntax Constraints

The syntax constraints as defined in the specification of the modularisation language for both encapsulation and interfacing concepts apply when realising this extension based on ADOxx. Listing 8.4 introduces the implementation of the cyclic fragment nestings constraint check (*C5*). The check ensures that a fragment cannot directly or indirectly nest itself. Basically, this check makes sure that the graph represented by the fragments as nodes and nested relation as edges is a directed, acyclic graph (DAG).

```
boolean hasCyclicNestings (IAdoCoreFragment aRoot,
    IAdoCoreFragment aFragment)
{
  Iterator<IAdoCoreFragment> aNestedFragments = aFragment.
    getNestedFragments();
  while (aNestedFragments.hasNext())
  {
    IAdoCoreFragment aNested = aNestedFragments.next();
    if (aRoot == aNested || hasCyclicNestings (aRoot, aNested
    ))
    {
      return true;
    }
  }
  return false;
}
```

Listing 8.4: An example of syntax constraint of the ADOxx metamodel modularisation language: Cyclic nestings check

On the other side, Listing 8.5 shows a representative constraint check for the interfacing constructs. The check restricts illegal assignment of con-

crete attribute elements to an attribute interface construct (*C14*). Clearly, although allowed given the syntax schema, an interface concept, in particular the class interface, cannot contain member implementations but only interfaces, in our case, attribute interfaces.

```
boolean hasAttributeCheck (IAdoCoreClassInterface aCInterface
    )
{
  Iterator<IAdoCoreAbstractAttribute> aMembers = aCInterface.
    getAttrDefs();
  while (aMembers.hasNext())
  {
    IAdoCoreAbstractAttribute aMember = aMembers.next();
    //check the type of the member: attribute or interface
    if(aMember.type == IAdoCoreAbstractAttribute.
   TYPE_ATTRIBUTE)
    {
      return true;
    }
  }
  return false;
}
```

Listing 8.5: An example of syntax constraint of the ADOxx metamodel modularisation language: Attribute interface aggregation check

8.2.2 Extending the Semantics

Being a supporting construct that helps organising metamodel parts into reusable units, neither fragment nor interface constructs have instance counterpart concepts on the model level, as this is usual for core metamodelling constructs. Hence, model-level semantics is neither affected nor extended. On the other side, the modularisation language brings new semantics on the metamodel level as we specified in the previous chapter.

Static Semantics

The language for metamodel modularisation introduce a number of static semantics definitions as specified previously. Listing 8.6 exemplifies the implementation of the few of those definitions as an extension of ADOxx. The first function declares the top fragment to be a fragment that doesn't have an owner fragment (*Definition 7*). An ownerFragment is an inverse property on the element side of the relation *nestedElements*. The second method (Line 6) defines an independent fragment to be a fragment that doesn't contain any dependent fragments by querying the amount of dependent objects following the relation *dependentFragments* (*Definition 11*).

```
boolean isTopFragment (IAdoCoreFragment aFragment)
{
  return  (aFragment.ownerFragment == null);
}

boolean isIndependentFragment (IAdoCoreFragment aFragment)
{
  return  (aFragment.dependentFragments.count() == 0);
}
```

Listing 8.6: Examples of static semantics implementation of the ADOxx metamodel modularisation language: TopFragment and IndependentFragment

Dynamic Semantics

The dynamic semantics of the modularisation constructs and its realisation in ADOxx may be explicated on the example of the dynamic semantics rule for the deletion of nested fragments (*Definition 18*). Basically, if a compound fragment is deleted, all nested elements must also be deleted. Listing 8.7 illustrates the implementation of this rule. On deleting a fragment, the function will iterate on nested fragments and recursively delete all nested occurrences (Line 9).

```
1  void deleteFragment ()
   {
3    Iterator<IAdoCoreFragment> aNestedFragments = this.
       getNestedFragments();
     while (aNestedFragments.hasNext())
5    {
       IAdoCoreFragment aNested = aNestedFragments.next();
7
       //recursively trigger deletion of nested fragments
9      aNested.deleteFragment ();
       this.nestedFragments.remove(aNested.ID);
11   }
   }
```

Listing 8.7: Example of dynamic semantics implementation of the ADOxx metamodel modularisation language: Deletion of nested fragments

8.2.3 Extending the Notation

The notation of the metamodel modularisation language extends the ADOxx tree-based and form-based notation for visualising and editing the constructs such as fragment, class interface and attribute interface. Hence, for each concrete metamodel element introduced a form-based syntax is introduced to provide the possibility to edit their respective properties. In addition, a

tree-based syntax is desirable for visualising the fragment nesting hierarchy and fragment dependencies.

8.3 Implementing Metamodel Composition in ADOxx

This section deals with the implementation of the metamodel composition language in ADOxx. As defined in Chapter 7, the metamodel composition language contributes to MMEL by providing a rich set of well-defined composition operators for black-box, grey-box and white-box metamodel composition. Recalling the supporting metamodelling capabilities of the ADOxx metamodelling language as discussed in Chapter 5, ADOxx supports white-box composition operators such as class inheritance and aggregation. In the following, we show how the syntax, the semantics and the notation of the ADOxx Meta^2-Model may be extended to allow for advanced metamodel composition.

8.3.1 Extending the Syntax

The metamodel composition language introduces several important composition operators on top of the standard operators already existing in ADOxx. The composition extension builds on top of both the core metamodelling constructs of ADOxx and the modularisation constructs introduced earlier. Similar to the implementation of the modularisation extension, for the sake of simplicity, we restrict the implementation of composition operators in this work to those dealing with class constructs and attribute constructs, and advise the realisation of the operators for other metaclasses to be done in similar fashion.

Syntax Schema

Black-Box Composition The key operators of the black-box metamodel composition are *interface realisation* and *interface subtyping*. In general, interface realisation operator binds the provided and required interfaces of metamodel fragments. In particular, it specifies how a concrete element realises a required interface. The ADOxx syntax schema is extended by the interface realisation on the class interface level (*classInterfaceRealisation*) and on the attribute interface level (*attributeInterfaceRealisation*), by stating that the core class realises its interface counterpart, so as the attribute its attribute interface, respectively (see Figure 8.4). A core class may realise many class interfaces, and vice versa, an interface may be realised by many classes. The same applies for attributes. Further, interface subtyping allows for creating type hierarchies, which implies compatibility and extension on the interface level. We extend the ADOxx syntax schema by adding the

interface subtyping for both class interfaces (*classInterfaceSubtyping*) and attribute interfaces (*attributeInterfaceSubtyping*).

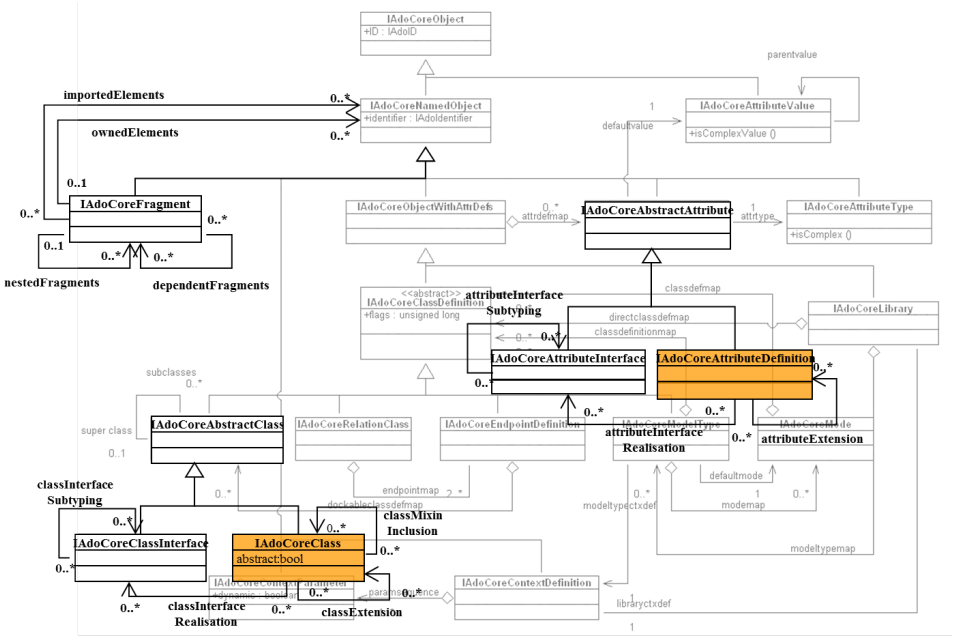


Figure 8.4: Extending the ADOxx Meta^2-Model for metamodel composition

Grey-Box Composition Grey-box composition introduces a novel composition operator for injective element extension based on elements implicit interfaces. For the extension operator to work, a base element, a extender element and the extends relation are needed. Hence, we extend the ADOxx syntax schema by adding the corresponding extension relation for classes (*classExtension*) and attributes (*attributeExtension*) (see Figure 8.4). For an element to play the role of an extender, it must be declared as abstract. In ADOxx, the property abstract is available for classes, but it is missing for attributes. Thus, we add this property as well with the same semantics that an abstract attribute cannot be instantiated. Having an extender element for classes means that extension points are the attribute and the class interface sets. For an attribute, this is the attribute interface set. Finally, the class extender, as a type, is an abstract metaclass, not having an instance counterpart on the model level. It is solely used for composition purposes on the metamodel level.

White-Box Composition White-box composition extends the standard white-box composition operators of ADOxx with the new operator for *mixin-based* composition. Mixins overcome the problem of multiple class inher-

itance allowing to mixin classes orthogonal to the inheritance hierarchy. Therefore, we extend the ADOxx syntax schema by adding a new mixin inclusion relation on the class level (*classMixinInclusion*) (see Figure 8.4). Mixin operator states that a core class may mixin other classes (which are declared abstract) and include their structural features (attributes) in its definition. While the same metaclass restriction automatically applies (mixin inclusion only between classes), ADOxx introduces an extension to this restriction on the metaclass property level. For the class mixins, the parent class must adhere to the mixin class with regard to the metaclass property "repository class"[2]. For example, the class *VersionableMixin* has a metaclass property to be a *repository* class. In that case, the class *Document* can only mix in the class *VersionableMixin*, if it is a repository-based class, too. Finally, note that for the sake of simplicity, we do not define the mixin inclusion feature for attributes.

Syntax Constraints

The syntax constraints defined in the specification of the composition language are applicable when realising this language extension in ADOxx. Constraints such as avoiding acyclic interface subtyping (*C13*), acyclic extension (*C60*) and acyclic mixin inclusion (*C80*) may be implemented in the similar way as we exemplified for the acyclic inheritance constraint earlier. Nevertheless, metamodel composition syntax introduces other constraints which implementation is worth mentioning. Listing 8.8 shows the implementation of the constraint *C61* that restricts that an extender class must be declared abstract. The same check may be used to implement the restriction *C82* for a valid mixin class.

```
boolean isValidClassExtender (IAdoCoreClass aExtender)
{
  return aExtender.isAbstract();
}
```

Listing 8.8: An example of syntax constraint of the ADOxx metamodel composition language: Valid extender class

8.3.2 Extending the Semantics

Akin to the modularisation language, the semantics of the metamodel composition language are realised on the metamodel level, this being the language's application domain. Although constructs such as interface and interface realisation have indirect implications on the model level, they do

[2]Repository class is a special built-in feature of the ADOxx metamodelling language, that enables instances of classes to be cross-model reusable repository objects instead of existing only inside of a model scope.

not imply any modifications in the mechanisms on the model level[3]. In the following, we explicate the realisation of some of the semantic rules for metamodel composition in ADOxx.

Static Semantics

The semantics of the black-box composition operator *classInterfaceRealisation* is, by far, one of the key semantic extensions for the black-box metamodel composition. A class is valid realisation of a class interface only if it realises all attribute interfaces of that class interface (*Definition 21*). Listing 8.9 unfolds its realisation. The method realises a query which checks, for a given core class and a class interface, whether the class is a valid realisation of the class interface. Starting with line 4, for each attribute interface, we check whether there is an attribute of the class, that implements that interface (from line 10).

```
boolean isRealisationClass (IAdoCoreClass aClass,
    IAdoCoreClassInterface aCInterface)
{
  Iterator<IAdoCoreAttributeInterface> aAIfaces = aCInterface
    .getAttributes();
  while (aAIfaces.hasNext())
  {
    IAdoCoreAttributeInterface aAInterface = aAIfaces.next();
    boolean bHasRealisation = false;

    Iterator<IAdoCoreAttributeDefinition> aAttributes =
    aClass.getAttributes();
    while (aAttributes.hasNext() && !bHasRealisation)
    {
      IAdoCoreAttributeDefinition aAttr = aAttributes.next();
      //checks if the attribute implements the interface
      bHasRealisation = aAttr.hasInterface (aAInterface);
    }
    if (!bHasRealisation)
    {
      return false;
    }
  }
  return true;
}
```

Listing 8.9: Example of static semantics implementation of the ADOxx metamodel composition language: Class interface realisation

The method will return false as soon as it detects an attribute interface

[3]Note that this doesn't mean that, if appropriate, one couldn't define additional semantics on the model level later on. Similar to the inheritance, which has dual semantics as an attribute reuse mechanism on the metamodel-level and as an object polymorphism mechanism on the model level, one could think of some advanced model-level semantics for other compositional relations, too.

which doesn't have an appropriate implementation. Otherwise, the class is valid realisation of the class interface.

Another very important semantic rule, which is the production rule of the class interface realisation relation and the allowed endpoint targets relation, is the derivation of endpoint targets (*Definition 32*). To recall, if a class interface is a target of some concrete endpoint, then all realisations of that interface are valid targets of that endpoint. This rule leverages the interface notion to allow for flexible black-box composition, as it introduces a level of indirection when connecting classes over endpoints. Instead of specifying the concrete class as an allowed target of an endpoint, we place the interface as a target, thus allowing any other class that implements that interface to be allowed target of that endpoint. Listing 8.10 shows the realisation of this semantic rule.

```
boolean isAllowedEndpointTarget (IAdoCoreEndpointDefinition
    aEndpoint, IAdoCoreClass aClass)
{
  Iterator<IAdoCoreAbstractClass> aTargets = aEndpoint.
    getTargets();
  while (aTargets.hasNext())
  {
    IAdoCoreAbstractClass aTarget = aTargets.next();

    if (aTarget == aClass ||
        (aTarget instanceof IAdoCoreClassInterface &&
            aClass.hasInterface(aTarget)))
    {
      return true;
    }
  }
  return false;
}
```

Listing 8.10: Example of static semantics implementation of the ADOxx metamodel composition language: Derived endpoint targets

Dynamic Semantics

The dynamic semantics of the composition operators and its realisation in ADOxx may be explicated the best by the example of the *extension operator* for grey-box composition. In particular, the *classExtension* relation implies that if the class extender extends the base class, the attributes of the extender are injected/assigned to the base class (*Definition 46*). Listing 8.11 unfolds the implementation of the grey-box class extension in ADOxx. As one can see in line 7, although the extension is syntactically non-invasive on the side of the base class (the extender declares which class it extends), it is realised invasively based on the atomic attribute aggregation relation. This kind of element injection based on implicit extension points is the cornerstone of the dynamic invasive grey-box composition.

```
void extendClass (IAdoCoreClass aBase, IAdoCoreClass
    aExtender)
{
  Iterator<IAdoCoreAttributeDefinition> aAttributes =
    aExtender.getAttributes();
  while (aAttributes.hasNext())
  {
    IAdoCoreAttributeDefinition aAttr = aAttributes.next();
    aBase.attrDefMap.add (aAttr);
  }
}
```

Listing 8.11: Example of dynamic semantics implementation of the ADOxx metamodel composition language: Grey-box class extension

Regarding the mixin inclusion composition operator, the inclusion of attribute elements of the mixin class to the base class is realised similarly (*Definition 59*). Referring to the same Listing 8.11 if we replace the second input parameter to accept the core classes (as mixins) in Line 1, then the same method may be used to realise the mixin-based composition. Clearly, the *classMixinInclusion* composition relation must exist between the two input classes.

8.3.3 Extending the Notation

The notation of the metamodel composition language extends the ADOxx tree-based and form-based notation for visualising and editing the composition constructs. In particular, the form-based syntax for class, class interface, attribute and attribute interface can be extended to support the definition of corresponding interface realisation and subtyping relations. Furthermore, the form for the class definition can be extended with options to define class extensions and mixin inclusions.

8.4 Applying MMEL Towards Modular Modelling Methods in ADOxx

In this section we discuss how MMEL, the cornerstone of the MME approach, contributes to the realisation of metamodel and functionality building blocks (MFBs), a more general approach to modular modelling method engineering in ADOxx. The MFB concept itself relies on the idea of method fragments as introduced by [Kühn, 2004] (see Chapter 2).

8.4.1 Metamodel and Functionality Building Blocks (MFBs)

In Chapter 2 we discussed the basic notions of modelling methods. To recall, a modelling method is a triple of a modelling language, mechanisms and a process. In order to support method integration based on reusable

fragments, the notion of method fragments has been introduced [Kühn, 2004] (see Section 2.2). Method fragments encapsulate single method elements and allow for their composition. Metamodelling platforms such as ADOxx are tools to realise modelling languages and methods (see Chapter 4.2). The realisation counterpart of the method fragment concept in ADOxx is called *Metamodel and Functionality Building Block (MFB)*. In the following, we introduce the notion of an MFB and discuss its particularities in the context of the modularisation and composition.

The Notion of an MFB

An MFB is a reusable modelling method building block that encapsulates the *metamodel* and/or *functionality* part of a modelling method. The metamodel part called *metamodel block (MB)* is here used in a broader sense to represent the static, structural part of the modelling method. The *functionality block (FB)* refers broadly to the dynamic part of the method, which may be manifested as modelling language dynamic semantics, algorithms and mechanisms, or even functionality referring to the process guidance. Following the definition of a method fragment, an MFB may be *atomic* or *composite*. A composite MFB consists of atomic and other composite MFBs. If an MFB is atomic, it represents only one of the method elements (metamodel or functionality). By virtue of composition, complex MFBs may be defined. Regardless of the type, an MFB may contain a concrete implementation or an interface that exposes the implementation, or both. Figure 8.5 illustrates a high-level metamodel of the MFB concept.

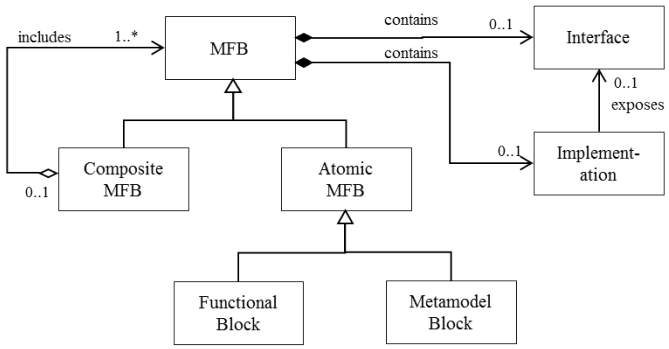


Figure 8.5: The metamodel of an MFB

The compositionality of atomic MFBs into composite blocks that contain both the metamodel and the corresponding functionality is one the key characteristics of the MFB concept. This way, construction of modelling methods is raised to the level of combining congruent, self-contained, reusable method fragments.

Types of MFBs

MFBs may be categorised according to the type of a method element they represent. By combining such MFBs additional composite types arise. Furthermore, MFBs may vary according to the abstraction level.

The following types may be identified based on the *compositionality* characteristic:

- *Atomic MB.* An atomic MB consists solely of a metamodel building block. Its content is represented by a set of metamodel elements defining some logically meaningful fragment of a metamodel. Such fragment does not include/depends on any other fragments. For example, a Petri Net MB consists of metamodel elements such as Transition, Places and Arcs.

- *Composite MB.* Composite MBs combine atomic or other composite MBs together. For example, a Coloured Petri Net may be a composite MB that combines the atomic Petri Net MB and the Petri Net Colouring extension MB.

- *Atomic FB.* Likewise, an atomic FB consists solely of the functional part. Since FBs do not require any metamodel part, the functionality of an FB is applicable for any metamodel. That said, an FB usually implements generic algorithms and mechanisms that are based on meta-metamodel level concepts (see Section 2.1.4 in Chapter 2). An example of an FB may be a simple Object Counting algorithm, that is applicable on objects of any class.

- *Composite FB.* Composite FBs combine atomic and other composite FBs to create complex functionality. For example, A Model Report FB may use the Object Counting FB to generate a report about the number of objects per class for a given model.

- *Composite MFB.* Composite MFBs are composites of at least one metamodel block and one functionality block. When used as extension blocks, a complete MFB extends the base for both the metamodel and functionality.

Furthermore, according to the *abstraction* characteristic, MFBs vary based on the fact whether they contain only the implementation, only the interfaces, or both. The following types exist:

- *Concrete MB.* A concrete MB represents an instantiable metamodel fragment. An example of a concrete MB is a BPMN process metamodel consisting purely of implementation elements.

- *Abstract MB.* Abstract MBs or (M)Bs are non-instantiable metamodel fragments. A MB is abstract, if it contains at least one abstract element not having a concrete implementation. An example of a generic MB is an abstract process metamodel consisting purely of interface elements for process modelling, that specify what a concrete MB needs to provide in order to support, i.e. realise the abstract process modelling specification.

- *Concrete FB.* Concrete FBs provide concrete implementation of a certain functionality. The functionality may be generic, applicable for any MB or or specific to a certain MB.

- *Abstract FB.* Abstract FBs or (F)Bs are functional building blocks without implementations. Usually they define a set of interfaces that are implemented by concrete FBs.

- *Concrete MFB.* Concrete MFBs are composites of at least one concrete metamodel block and one concrete functionality block. The functionality is written usually specific to the metamodel, thus corresponding to (metamodel-)specific algorithms and mechanisms. Usually, the functional part represents the implementation of the language semantics. An example of a complete MFB may be a combination of the Petri Net MB and a Reachability FB that validates the reachability of states in petri net models.

- *Abstract MFB.* An abstract MFB or a generic, partial MFB contains both MB and FB, but at least of the blocks is of abstract type such as (M)FB, M(F)B, or (MF)B. Usually, such MFB contains only generic MB and a concrete FB that is based on it. Such MFB may be considered as a hybrid, adaptable mechanism (see Chapter 2.1.4 for definitions). Furthermore, partial MFBs cannot be used out-of-the-box, since they need an adequate parameterisation of the abstract part. An example of a partial MFB is a simulation algorithm that is based on a generic, parameterisible process metamodel.

Composition of MFBs

When combining different types of MFBs, some basic and complex composition operations may be identified. *Binding*, *extension* and *grounding* are, what we call, the basic composition operations for MFBs.

- *Binding.* Binding is about connecting the abstract MFB to its implementation MFB of the same type. For example, a concrete MB realises the abstract MB by binding its concrete elements to the interface counterparts in the (M)B. The same applies for the functionality blocks.

- *Extension.* Extension is any kind of composition operation of extensional character that connects two blocks of the same type. It may be based on composition operators such as inheritance, aggregation, extension, mixin inclusion, etc. It applies to any combination of concrete or abstract MBs or FBs.

- *Grounding.* Grounding is a composition operation that connects the functionality block to the corresponding metamodel block. It applies for any combination of concrete or abstract FBs and MBs, as long as the source block is an FB and the target block of grounding is an MB.

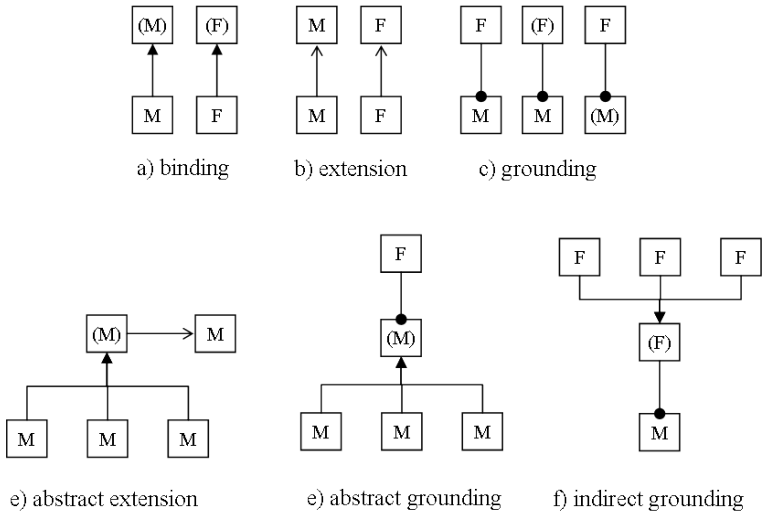


Figure 8.6: Variability of MFB composition operations

Having the basic MFB composition operations, complex compositional patterns may be applied to allow for generality, flexibility, substitutability when creating modular modelling methods. Without extensive elaboration on the variability of such complex combinations, we explicate the idea based on the following patterns such as, abstract extension, abstract grounding and indirect grounding.

- *Abstract extension.* Abstract extension is basically an abstract MFB that extends another MFB. While being abstract, many different concrete MFBs may be concrete extensions as long as they are correctly bound to the abstract MFB. In that case, the abstract MFB prescribes what are the allowed extension points, which concrete MFBs can use to extend the base MFB.

- *Abstract grounding.* In order to allow for a flexible functional block applicable for a family of similar metamodels, an FB may be grounded

to an abstract metamodel block (M)B. This way, any concrete MB that has a valid binding to the (M)B is a grounding candidate for that FB. A simulation algorithm applicable for various process models is a good example for abstract grounding.

- *Indirect grounding.* On the other side, there may be different variants of the same functionality that should be grounded on one particular metamodel. This can be done by binding the abstract functional block (F)B to an MB, and by letting concrete FBs bind to the (F)B. An example of this may be a petri net language with an abstract reachability algorithm, having different implementations focusing on different optimisation techniques.

Figure 8.6 illustrates the variability of the basic and complex MFB composition operations.

8.4.2 Realisation of MFBs in ADOxx

Concepts for modularisation and composition of both metamodel and functionality are crucial for the realisation of MFBs. Such concepts should be an integral part of the corresponding formalisms that are available for the implementation of the metamodels and of the functionality. As discussed earlier (see corresponding sections in Chapter 4 and Chapter 8), ADOxx provides the ADOxx Meta^2-Model, a metamodelling language for the metamodel definition. Likewise, ADOxx provides a set of APIs that are embedded in the scripting language such as JavaScript or in the GPLs such as Java and C++, that are used to implement method functionality, i.e., to realise dynamic semantics of modelling languages and corresponding algorithms and mechanisms of a modelling method. In the following, we discuss the modularisation and the composition concepts of those formalisms in ADOxx. In particular, we discuss how the MME approach and the language MMEL contribute to the modular and compositional capabilities of the metamodel part of the MFB concept in ADOxx.

Modularisation and Composition of Functionality Blocks

Since ADOxx offers its APIs through standard scripting and programming languages, it inherently provides adequate constructs for both modularisation and composition of the method functionality. Depending on the programming language applied, notions such as namespaces, packages, access modifiers and interfaces may be used to organise program code into reusable components. In addition MFB-specific (metaprogramming) APIs are provided that allow for accessing metainformation about MFBs such as its dependencies, properties, availability, etc. Similarly, composition concepts

may be used to compose FBs. These include both standard "push" techniques via API calls between FBs, but also "pull" programming models such as callbacks, event-based composition and messaging[4]. The latter two are especially useful when composing prefabricated black-box components.

Modularisation and Composition of Metamodel Blocks based on MMEL

The modularisation and the composition of metamodel blocks in ADOxx is where the MME approach is applied. In particular, the ADOxx Meta^2-Model is extended for the modularisation and composition concepts of MMEL as described in Chapter 8. In particular, the notion of an MB is realised using the *metamodel modularisation language*. Metamodels are encapsulated in fragments using the constructs defined in the *metamodel encapsulation language*. The *interfacing language* is used to define explicit interfaces of MBs. Clearly, if an MB is a concrete MB, the metamodel fragment may consists of concrete metamodel elements, but also define a set of corresponding interfaces other MFBs may use. In case of abstract MBs, the metamodel fragment will consist of interfaces or abstract elements that do not have a corresponding implementation in that fragment. The notion of nesting is used to define internal fragments of an MB, not visible to the outside. Dependencies declare to which fragments of other MBs, a fragment of a particular MB depends on, for composition purposes. On the other side, the *metamodel composition language* provides standard and advanced composition operators for *black-box, grey-box and white-box composition* of MBs. The *binding composition* of abstract MBs and concrete MBs may be done using the interface realisation composition operator. Furthermore, different kinds of *extensional compositions* between MBs are possible using black-box interface subtyping, grey-box extension operator, as well as using appropriate white-box operators such as aggregation and mixin inclusion.

Grounding: Composition of Functionality Blocks and Metamodel Blocks

Modularisation of MBs into self-contained fragments based on MMEL allows for combining and deploying MBs together with a corresponding FB to form composite reusable units available for further combination. The FB part of such MFB *grounds* on the concepts from the MB to build up a particular functionality specific to that metamodel. Since ADOxx provides dedicated APIs to access metamodel information, and by that, also the APIs related to the constructs of the MME language, the composition of FBs and MBs

[4]In the context of component composition, Szyperski [Szyperski, 2002] defines standard method calls as being traditional "pull" programming models, whereas event-based notifications and messaging are referred to as "push" programming models.

is based on the standard component API calls from the functional block to the metamodel block. Note that the metamodel block may be concrete or abstract, thus allowing for the realisation of *abstract and indirect groundings* and other complex MFB compositions.

8.5 Chapter Summary

In this chapter we elaborated on a possible implementation of MMEL, a language for modular metamodel definition, within the metamodelling platform ADOxx. In particular, we demonstrated how the ADOxx metamodelling language ADOxx Meta^2-Model is extended by MMEL on all three language aspects, the abstract syntax, the semantics and the notation, in order to support metamodel modularisation (Section 8.2) and metamodel composition (Section 8.3). With respect to the requirements for modular metamodel engineering, the implementation fulfilled all of them successfully. The core metamodelling language has been natively extended by the modularisation and the composition concepts (RT1). As a consequence of the native language extension, the ADOxx metamodelling core has been extended to support interpretative composition derivation (RT2), i.e. no transformation/code generation step is needed when combining metamodel fragments. While core and modular constructs reside on the same abstraction level, generative techniques become superfluous. Finally, since the introduced modularisation and composition concepts belong solely to the supporting capabilities of metamodelling languages, they do not influence the core concepts and thus do not break compatibility with the underlying model-level mechanisms (RT3). Instead, these concepts extend ADOxx metamodelling language with advanced, metamodelling techniques that foster reuse and modular definition of metamodels. In addition, in Section 8.4, we discussed how MMEL fills in the missing piece of the puzzle for the realisation of the more general concept of MFBs in ADOxx towards modular engineering of modelling methods.

Part IV

Evaluation

Chapter 9

Case Studies for MMEL in OMILab

"We can't solve problems by using the same kind of thinking we used when we created them."

Albert Einstein

As we introduced the general concept of MME in Chapter 6, we mentioned that there is nothing that can be done with metamodel fragments, that cannot be done without them. While the end result of metamodelling is the same, the way how metamodels are developed differs dramatically. Instead of developing complex metamodels from scratch, MME allows for creating metamodels by combining existing metamodel fragments. Furthermore, it allows for flexible extensions of existing metamodels by pluging in metamodel add-ons. Systematic reuse, flexibility in adaptations, extensions and overall increased efficiency in developing metamodels are key benefits of MME. In this chapter, we aim at evaluating the usefulness and the applicability of MME, and in particular of its language MMEL, based on two case studies in modular construction of hybrid modelling languages in the context of OMILab[1]. The selection of modelling languages for systematic evaluation in both case studies reflects the major duality of modelling concerns (structure, behaviour) and the diversity of modelling domains in enterprise modelling (Business Processes (BP), Enterprise Architecture (EA)).

The chapter is organised as follows. In Section 9.1, we use modularisation and composition concepts of MME and constructs from MMEL to design the hybrid, enterprise modelling method BPMS. In Section 9.2, in the second case study, we demonstrate how MMEL can be applied to construct

[1]OMILab [OMI, 2015] is a research and experimental laboratory for the conceptualisation, development and deployment of modelling methods and the models designed with them.

the Hybrid PDDSL, a hybrid DSML that combines a language for network devices modelling with the ontology language OWL2 for consistent network device management. Section 9.3 summarises the chapter.

9.1 Case Study in BP: Modular BPMS

The Business Process Modelling Systems (BPMS) method is a modelling method used for enterprise-wide business modelling of four core enterprise areas business processes, products, organisations and information technology [Bayer and Kühn, 2013]. It is part of the more general BPMS framework for enterprise business process management [Karagiannis, 1995] and its successor methodology PMLC [Bayer and Kühn, 2013]. The BPMS method is the core modelling formalism used in the business process modelling tool ADONIS [Junginger et al., 2000, BOC, 2015] based on ADOxx.

In this section, we discuss a possible realisation of the metamodel part of the BPMS method using the modularisation and composition concepts and constructs of MMEL. To do that, we first need to introduce particularities of ADOxx metamodels in general before we continue with BPMS-specific application of MME.

9.1.1 Particularities of ADOxx Metamodels

A metamodel in ADOxx is a pivotal element for the definition of a modelling language and indirectly of a modelling method. Besides its core purpose of defining the abstract syntax of a modelling language, the ADOxx metamodel also integrates the definition of notation for the abstract syntax constructs. Furthermore, it represents the common data structure of the various system-specific and metamodel-specific algorithms and mechanisms being part of a modelling method. That said, such metamodel may grow big. Real-world, industry-relevant modelling methods contain metamodels having several thousands of metamodel elements[2], which make them complex, monolithic design and implementation artefacts that are hard to comprehend and maintain. Metamodel modularisation and composition techniques can help to tackle this complexity by dividing complex metamodels into smaller, self-contained, reusable fragments that can then be developed independently and flexibly recombined to create composite metamodels.

In order to be able to introduce modularisation of ADOxx metamodels, we need to understand what their basic constituents are. Basically, the content of an ADOxx metamodel is driven by the requirements from two sides, the domain of a modelling language, the functionalities applied.

[2]The metamodel size statistics stem from ADOxx-based projects at BOC. For example the standard BPMS method used for ADONIS tool has around 1300 defined metamodel elements, whereas the EAM metamodel of ADOit contains 900 metamodel elements.

That said, an ADOxx metamodel consists of *domain-driven elements*, and *functionality-driven elements*. Figure 9.1 illustrates the anatomy of ADOxx metamodels.

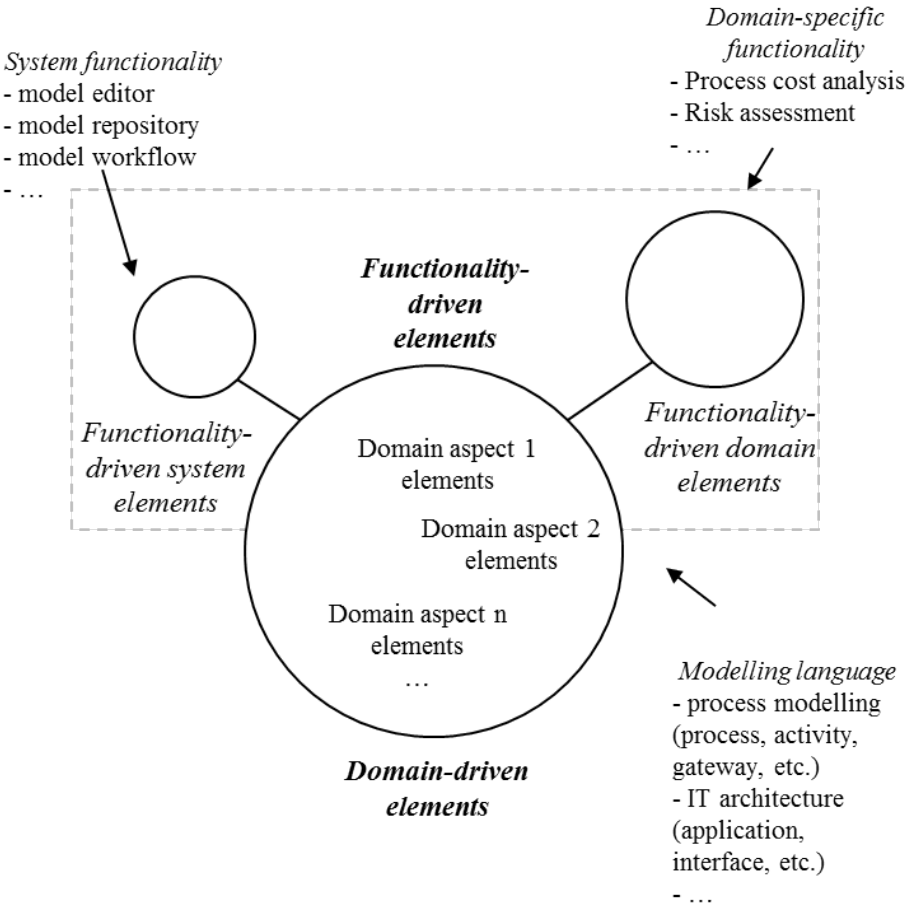


Figure 9.1: Anatomy of ADOxx metamodels

Domain-driven Elements

The metamodel elements required by the domain represent core constructs of a modelling language. Domain-driven, or domain elements are usually separated into different domain aspects, that represent single viewpoints of the system under consideration. For example, a class element Task is a domain element used to represent steps within a process. A Task may have some domain attributes such as Type and is related to other class elements such as events. Note, by defining such an element, we don't specify whether such element can be modelled in a graphical model editor or not.

Functionality-driven Elements

Functionality in a modelling method is applied on the model data, i.e. it requires certain data structures that are represented in the metamodel. Domain elements provide such structure, however additional elements may be needed, which we call, functionality-driven elements. There are two categories of functionality-driven elements:

- *Functionality-driven domain elements.* If a functionality-driven element extends the domain knowledge, it is considered domain-specific. For example, if a process model should support the functionality of process cost analysis, its tasks should contain properties to store costs. Clearly, there is no precise separation of domain-driven and functionality-driven domain elements, as one could argue that the latter is in fact a domain element. The difference lies in the purpose of the element, if it is used to model the core properties of an entity or to enable some domain-specific functionality.
- *Functionality-driven system elements.* On the other side, if metamodel elements are required by some system mechanisms, they are considered system-specific. For example, if the class Task should be modellable in the graphical editor, it needs to contain model editor (system) attributes, in order to persist two-dimensional position coordinates of task instances in a graphical process model.

9.1.2 Modularisation of the BPMS Metamodel

In the following, we illustrate how the metamodel of the BPMS method as a monolithic design artefact can be decomposed and then modularised into self-contained logical units (fragments) as a first step towards its modular design. In doing so, we apply the concepts of the metamodel modularisation language.

The BPMS Method

The BPMS method is a hybrid modelling method covering different enterprise areas and offering a variety of notations for enterprise modelling. The central part of the BPMS method is the standard language for business process modelling BPMN 2.0 [OMG, 2013]. As a hybrid method, BPMS extends BPMN with modelling languages for product, organisation and IT modelling as elaborated in [Rausch et al., 2011]. Figure 9.2 illustrates the hybrid usage of BPMN together with document models, organisation models and risk models.

Different notations in BPMS are represented by the ADOxx model type construct, representing a type of a diagram. The four areas of BPMS framework are thus represented by one or more model types. The model types and

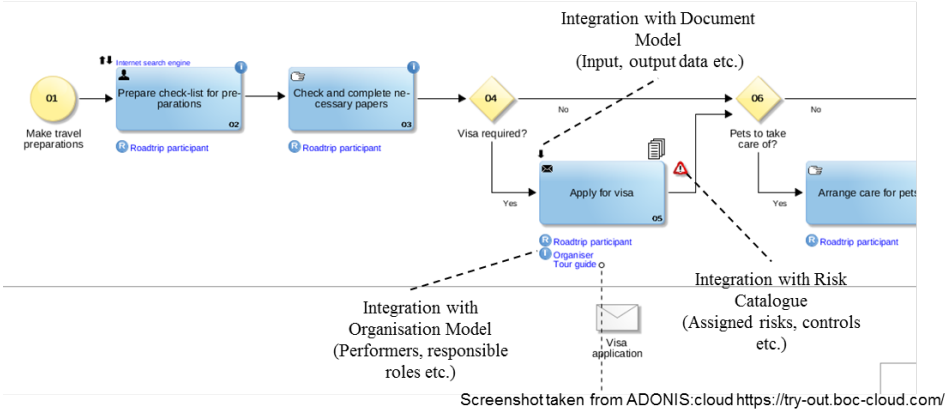


Figure 9.2: Hybrid usage of BPMN within the BPMS method

elements within it are interconnected to allow for hybrid, integrated modelling using the ADOxx interref concept. In the following, a short overview of the basic constructs of the BPMS method are given, to illustrate the size and the complexity of its corresponding metamodel. For detailed description, the interested reader should refer to [BOC, 2015, Bayer and Kühn, 2013].

Business Processes A model of a business process is the pivotal artefact of the BPMS method. To model process flows, BPMS method adopts the BPMN standard represented by the model type *Business Process Diagram*. A business process consists of a chain of atomic activities/tasks and subprocess calls. Start and end events, flow control gateways such as paralellities and decisions help to capture the process flow. Activities have connections to constructs from other aspects such as input and output documents, assigned organisational roles, etc. The model type *Process Landscape* allows for grouping and hierarchical structuring of processes as enterprise process building blocks. The integrated BPMN notation contributes with two additional model types *Conversation Diagram* and *Choreography Diagram* that are used to model interactions between business partners.

Products The model type *Product Model* allows for modelling of products and product components an enterprise has to offer. Products are connected to processes and organisation structures to depict which processes contribute to the production of products and by which organisational units.

Organisation The model type *Organisational Model* supports the modelling of organisational forms of the enterprise. Performer, Role, Organisational Unit are core constructs of this model type.

Information Technology For the modelling of the supporting IT, the model type *IT System Model* is available. It offers constructs such as Applications, Interfaces and Services to capture the high-level IT landscape. They help to document which applications and services exist in the enterprise, which interfaces they offer to support enterprise business processes, by whom they are used, and who is responsible for their maintenance. In addition, the *Use Case Diagram* is available (integrated from UML) to capture and identify users and their interactions with IT systems. In addition, the model type *Document Model* to model information objects such documents is used. The document class is, for example, also used to realise the document management system mechanisms. The detailed data structures about documents may be captured using the *Data Model*.

Risks and Controls The model types *Risk Catalogue* and *Control Catalogue* allow for documenting the enterprise internal controlling system (ICS). Risks represent events or developments that can negatively influence the fulfillment of enterprise goals. They may be assigned to processes and activities to help point out process fragments that may be negatively affected by such events. Controls may be assigned to risks to document procedures and rules that may prevent or minimise the occurrence of identified risks. Risks and controls may be seen as supporting constructs, i.e. add-ons for business processes to an achieve integrated process and risk management.

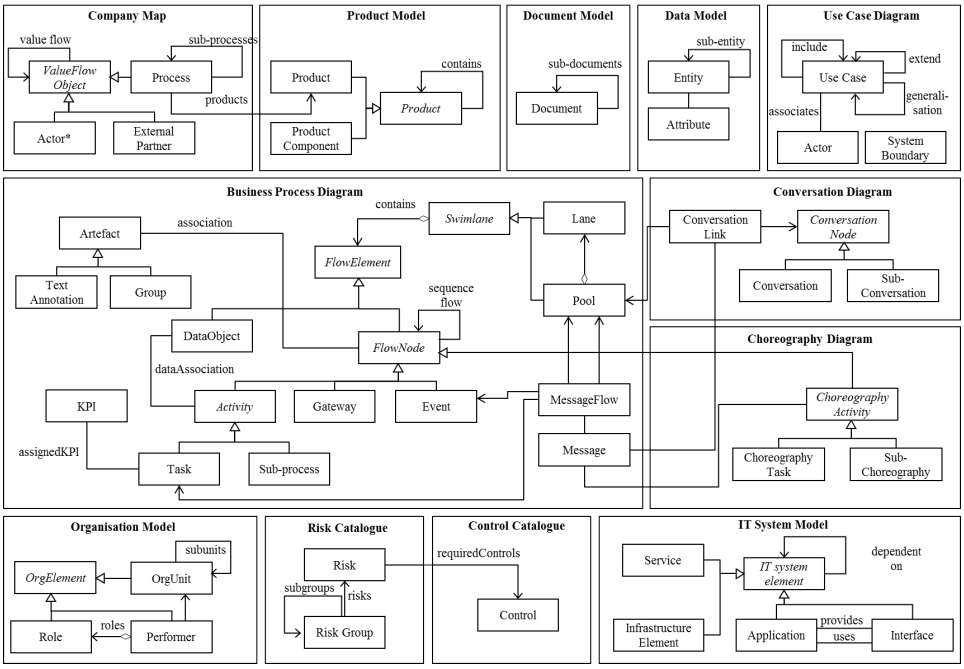


Figure 9.3: Simplified conceptual view of the BPMS metamodel divided into main model types

Figure 9.3 illustrates a significantly simplified conceptual view of the BPMS metamodel only considering some main classes and relations. Due to its size and complexity (approx. 1300 metamodel elements), the consideration of the cross-model references, as well as of the implementation view, i.e. of all domain and system elements of all types, i.e. incl. attributes, relation endpoints, etc. as well as system-relevant elements has been left out.

Modular BPMS: Identification of Metamodel Fragments

There may be different ways to decompose the BPMS metamodel, i.e. to identify self-contained metamodel modules, i.e. fragments. Considering the particularities of ADOxx metamodels, the natural way is to start the decomposition by dividing metamodel elements into *domain relevant* and *system relevant*. Within the BPMS domain, a good way for the categorisation of fragments is to follow the decomposition based on enterprise areas and single model types. On the other side, for the system elements, fragments are defined for each functional component that requires or provides corresponding metamodel elements. Note that system elements are domain-independent, thus they may be reused for other metamodels, too. Figure 9.4 illustrates a possible modularisation of the BPMS using the UML package diagram notation, including both domain-relevant and system-relevant fragments. Concrete fragments represented by packages are nested within high level fragments, where high level fragments serve as kind of grouping namespaces.

Since the introduction of each of the fragments would exceed the limits of this work, we explicate the metamodel modularisation based on one of the pivotal BPMS fragments, the *Business Process Diagram* fragment (BPMN fragment). BPMN does not provide sufficient support to model organisational structures [Zivkovic et al., 2007]. Thus, the BPMS method embeds the BPMN and extends it with languages for the comprehensive modelling of business processes and other enterprise areas such as organisation, products and IT [Rausch et al., 2011, Bayer and Kühn, 2013]. Modularising the core part of the BPMN for process modelling into a fragment with well-defined interfaces is desirable, in order to allow for its reusability via recombination within different hybrid modelling methods. BPMN constructs provide means for modelling of business process flows. Therefore, constructs such as the model type *BPDiagram* and the class *Task* are good candidates to expose them via interfaces such as model type interface *IProcessDiagram* and class interface *ITask*. Not only provided interfaces of the BPMN fragment are desirable, but also required interfaces from other fragments. Referring to the case study of extending the BPMN with constructs for organisation modelling from [Zivkovic et al., 2007] (see also Section 2.3), two explicit integration points seem appropriate. The class *Pool* may be regarded as a visual representation of participant, which cane business entities or roles. Hence, we introduce the class interface *IParticipant*, which is a required interface

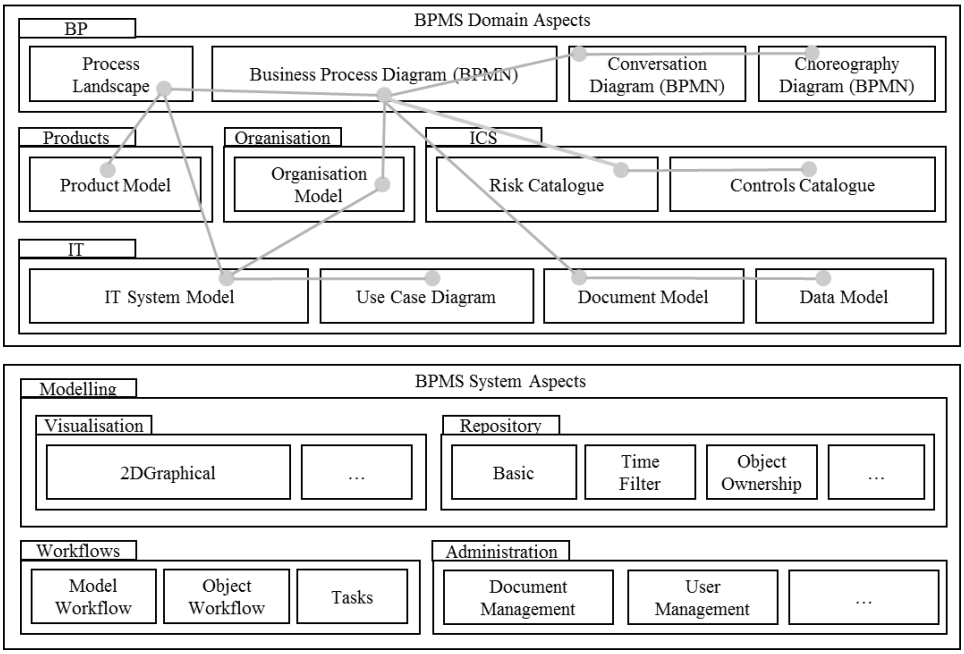


Figure 9.4: A possible modularisation of the BPMS metamodel (Connections between domain fragments represent major integration points.)

the should be realised by third fragments. Looking closer into the BPMN fragment elements, a further good candidate for a required interface is the concept of a task performer. Since performers of tasks may be different constructs, we specify the concept of performer as a class interface *IPerformer* and an extension point, such that third fragments can provide different implementations. Another extension point to consider in the BPMN fragment is the connection to risk management concepts, the modelling area not considered by the BPMN standard. Hence, we declare an additional required interface *IRisk* that specifies required structural features of the concept of risk. We connect the class *Task* to the risk interface via the relation *assigned risks*. Now, different risk concept implementations may be supported. Figure 9.5 shows the glass-box[3] view of the BPMN fragment.

9.1.3 Composition of BPMS Fragments (A Selection)

In the following, we demonstrate the application of the black-box, grey-box and white-box metamodel composition operators on the selected set of composition problems in the context of the modular BPMS.

[3]The glass-box view allows for the inspection of the internal elements of black-box fragments. In contrast to the white-box, it doesn't allow for any modifications.

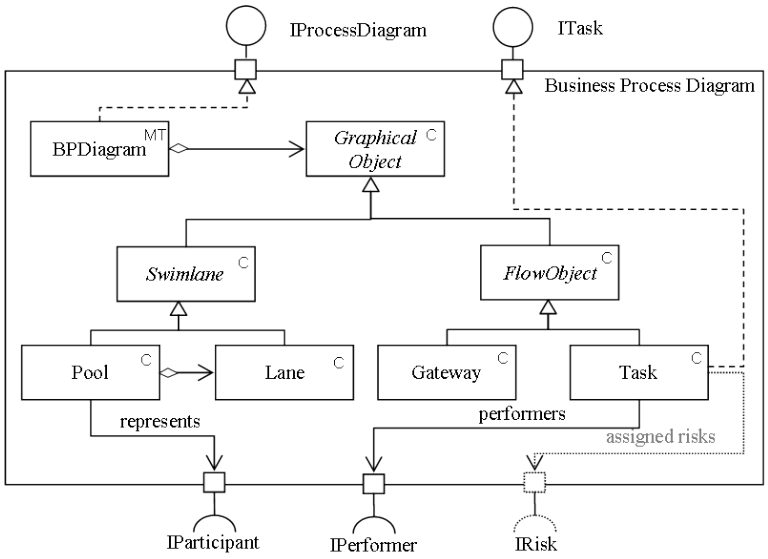


Figure 9.5: An example of the BPMS modularisation: BPMN Business Process Diagram Metamodel fragment (simplified metamodel)

Applying Black-Box Metamodel Composition

Having defined explicit black-box metamodel fragments, we can combine them using the composition operators for interface-based metamodel composition. We demonstrate how black-box composition may be applied by combining the fragment BPMN with fragments *Organisation Model (OM)* and *Risk Catalogue (RC)*. By doing so, we also explicate the modularisation of these two additional fragments. The OM fragment contributes the constructs for the modelling of organisational units, actual actors and their roles. We declare classes *OrganisationUnit* and *Role* to realise the interface *IParticipant* that are required by the BPMN fragment. This integration decision conforms to the integration points introduced in [Zivkovic et al., 2007]. We do this based on the *cInterfaceRealisation* composition operator. In the next step, the interface *IPerformer*, representing task performer role, is realised by organisational units (*OrgUnit*), roles (*Role*) and actors (*Actor*). In the following, we explain the application of the interface subtyping. We apply class interface subtyping to state the compatibility of the interfaces *IOrgElement* and *IPerformer*. This is motivated by the fact that fragment OM already provides the implementation for the interface *IOrgElement* based on the class *OrgElement*. Finally, we combine the fragments BPMN and RC by specifying the interface realisation between the class *Risk* and the interface *IRisk*. Figure 9.6 illustrates the introduced composition using the black-box composition operators.

We revisit the introduced black-box composition of BPMS fragments, in

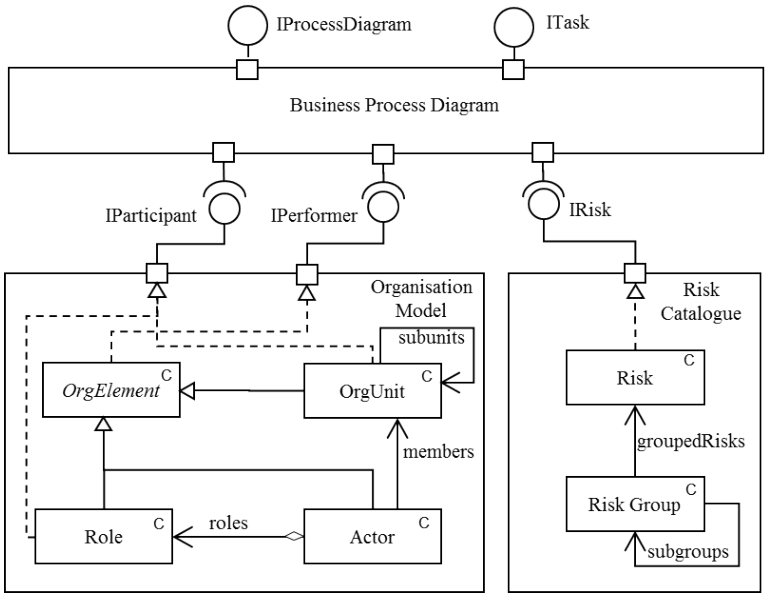


Figure 9.6: Applying black-box metamodel composition for BPMS: Composition of the BPMN fragment, OM fragment and RC fragment

order to explicate its implementation in more detail. In doing so, we use the notation of MMEL introduced in Chapter 3. This time, we describe the composition not only on the level of classes, but also on the level of attributes and relations with endpoints. Figure 9.7 illustrates the revisited fragment modularisation and composition solution. We define class *Task* as the realisation of the class interface *ITask*. In order to fulfill the contract of the provided interface, class *Task* aggregates the attribute *Name*, which in turn realises the attribute interface *IName*. Furthermore, BPMN fragment exposes one required interface, *IRisk*, without providing the realisation for it. Note how relation class *assignedRisks* connects the class *Task* with the interface *IRisk*. This is crucial for the flexibility of the solution. In doing so, any class that implements the interface *IRisk* becomes a valid target of the endpoint *ToAR*, i.e. of the corresponding relation class. On the other side, the fragment *RC* contributes with the class *Risk* that realises the interface *IRisk*. In order to do so, the fragment *RM* establishes the dependency to the fragment *BPMN* and imports the interface *IRisk* (marked with dashed line).

Applying Grey-Box Metamodel Composition

Grey-box metamodel composition allows for extending existing prefabricated fragments with extensions based on well-defined implicit interfaces. To demonstrate the grey-box composition, we will extend BPMS domain fragments such as the *BPMN* fragment and the *Process Landscape (PL)*

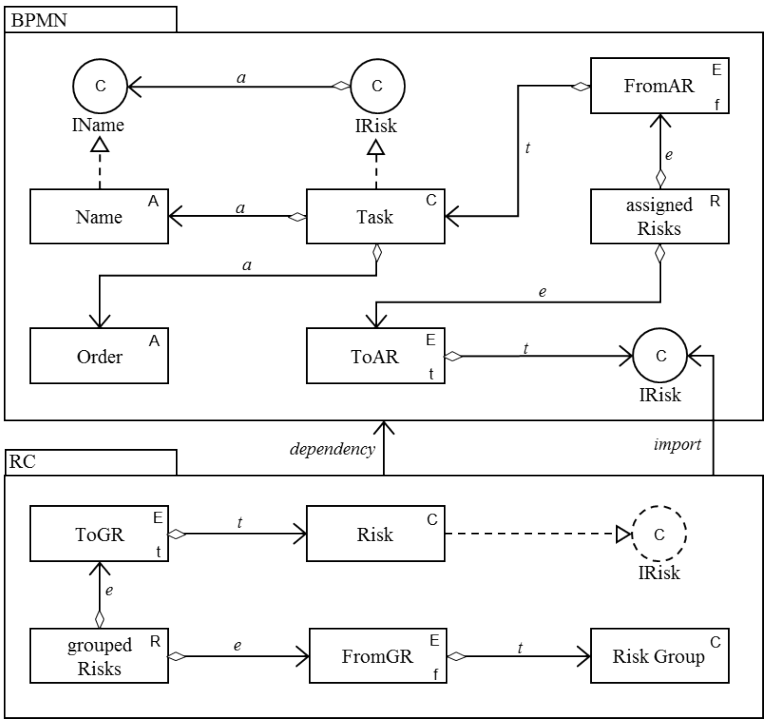


Figure 9.7: Realisation of black-box metamodel composition for BPMS: Detailed composition of BPMN and RC fragments

fragment with the functionality-driven, i.e. system fragment *Model Workflow (MW)*. Our goal is to allow for the versioning and state-based workflows of business process models within BPMS. Figure 9.8 illustrates the grey-box composition scenario.

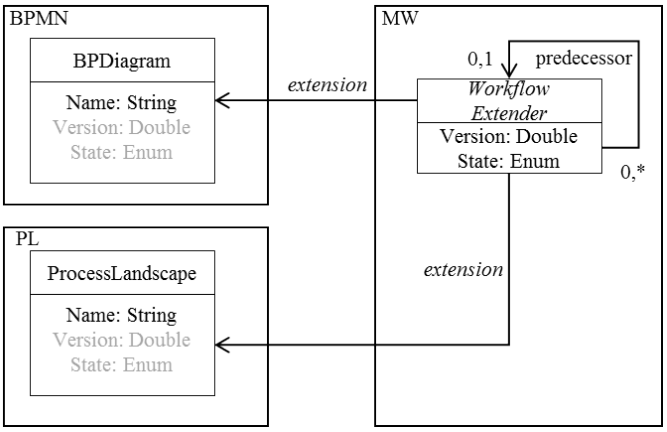


Figure 9.8: Applying grey-box metamodel composition for BPMS: Extending fragments BPMN and PL with extender fragment MF

The *MW* fragment is defined as an extender fragment. It contains the extender element for model types *WorkflowExtender* which contains two attributes relevant for the model workflow functionality such as *Version* and *State*. The version attribute is of type double and it stores incremental version numbers of models. The state attribute is an enumeration containing a predefined set of possible model workflow states (e.g. draft, review, released, rejected, archived). In addition, a predecessor relation is defined to establish predecessor-successor connections between different model versions that build up a model version tree. The actual metamodel composition relies on the *extension* composition operator. We declare that the *WorkflowExtender* extends model type elements *BPDiagram* and *ProcessLandscape* from the corresponding base fragments. This way, the model type attributes version and state will be injected to the corresponding base model types. Furthermore, by virtue of indirect relation end targets, these concrete model types will indirectly become targets of the predecessor relation. Figure 9.9 revisits the extension-basd grey-box composition scenario using the MMEL syntax and notation. Note the dependency and import directions. While extended fragments *BPMN* and *PL* remain unmodified and independent, the extender fragment *MF* encloses the semantics of the composition.

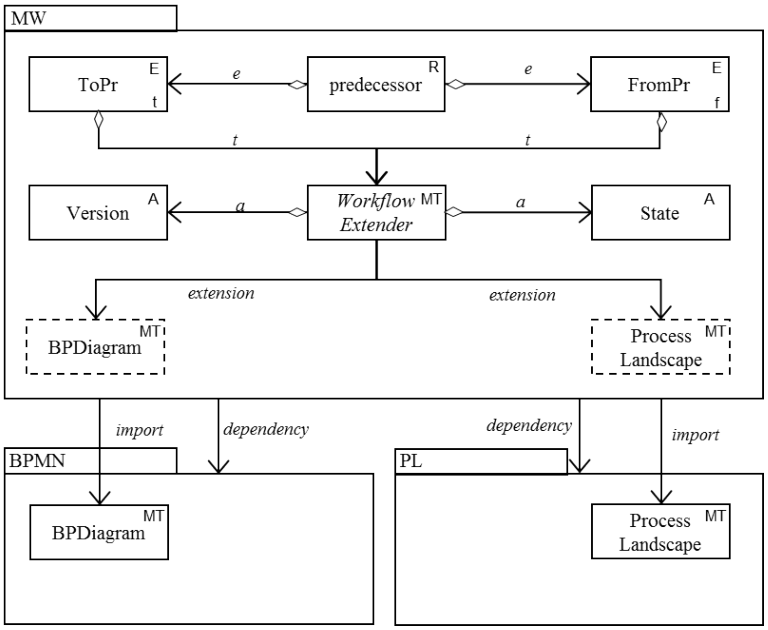


Figure 9.9: Extension-based grey-box metamodel composition for BPMS: Detailed composition of fragments BPMN and PL with extender fragment MF

Finally, if we think of an MFB that supports model workflows, the functionality block paired with the extender fragment may be deployed as a

reusable Model Workflow MFB in ADOxx. The abstract extender class is used as a kind of a stable interface against which the functionality may be coded and by which concrete model types may be retrieved.

Applying White-Box Metamodel Composition

White-box metamodel composition contributes to flexible modular metamodel definition with the *mixin inclusion composition* operator. We demonstrate the usage of the mixin inclusion operator in the context of BPMS by composing the domain fragments with system-relevant fragments in order to extend BPMS fragments with functional add-ons. Figure 9.10 shows the mixin-based white-box composition scenario.

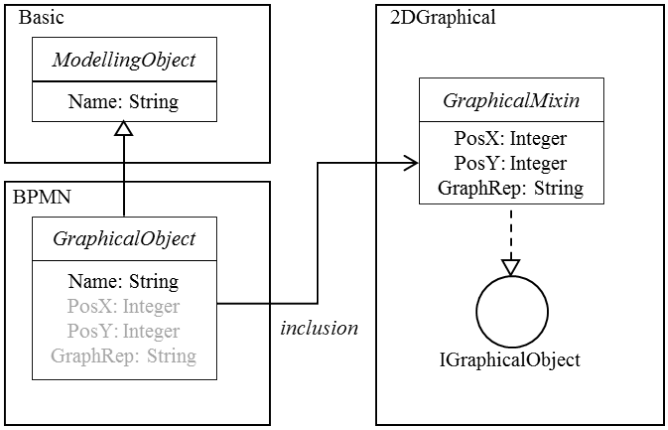


Figure 9.10: Applying white-box metamodel composition for BPMS: Mixin-based composition of the domain fragment BPMN and the system fragment 2DGraphical

Assuming that the graphical elements from BPMN (represented by the abstract class *GraphicalObject*) should be modelled using ADOxx graphical syntax in the graphical model editor, such a graphical element needs to fulfill certain metamodel requirements. These requirements are captured and encapsulated in the metamodel fragment *2DGraphical*. This fragment contains a mixin class *GraphicalMixin* which specifies that for a class to be used for graphical objects in a model editor, it needs to contain attributes for storing 2D coordinates of an object within a model (attributes *PosX*, *PosX*), as well as a graphical symbol definition (attribute *GraphRep*). Any class that mixes in the graphical mixin class will automatically fullfill the requirements by the ADOxx graphical model editor. Therefore, to perform the composition, we declare that the abstract class *GraphicalObject mixes in* the *GraphicalMixin* class. Note, by class mixing, the GraphicalObject class will include all structural features of the mixin class, i.e. it will also mix in the interface *IGraphicalObject*, thus automatically becoming a valid

realisation of that interface. Figure 9.11 illustrates the details of the aforementioned composition scenario using MMEL.

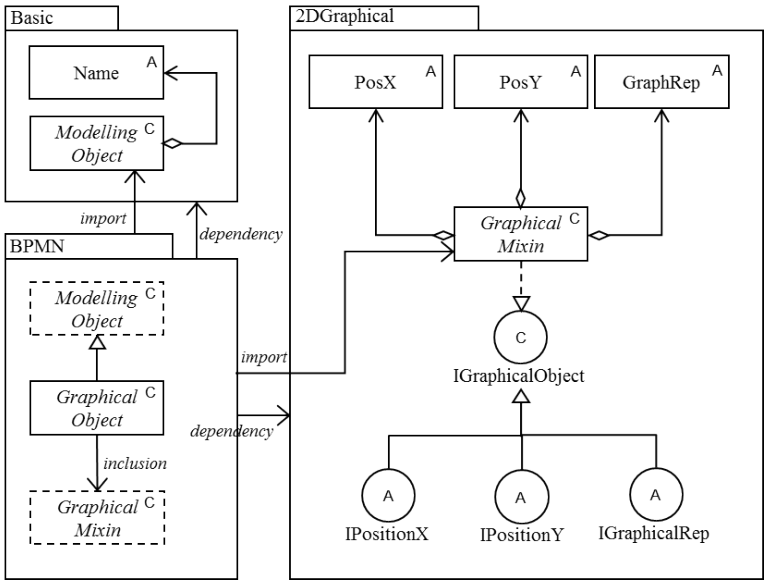


Figure 9.11: Mixin-based white-box metamodel composition for BPMS: Detailed composition of the domain fragment BPMN and the mixin fragment 2DGraphical

Finally, note that the mixin class *GraphicalMixin* may be combined by arbitrary classes. This also applies for the parent class *GraphicalObject* that may include arbitrary additional mixin classes. Also, since the mixin inclusion operator is orthogonal to the inheritance, the parent class can independently build inheritance hierarchies, while at the same time including mixin classes. This way, the mixin inclusion operator contributes to a more flexible metamodel composition.

Achieving Independent Fragment Composition

In Section 5.2, we discussed the characteristics of the non-invasive and invasive composition. We said that non-invasive composition does not modify the fragment being composed, but only the composer fragment is modified to capture the glue logic of composition. For example, while the black-box composition allows for the composition of prefabricated black-box fragments based on well-defined interfaces, the glue logic of the composition, i.e. the explicit dependencies are coded in one of the participating fragments.

However, the extension composition operator is a powerful composition construct that may be used to achieve the *independent fragment composition*. By introducing extender fragments that hold the glue logic of the composition, the extender fragment mixes in elements from one fragment and

injects them to the other fragment. Hence, the fragments to be composed remain unmodified and independent, while being composed by a third composition fragment. This kind of independent composition based on extender fragments may be applied in the context of both black-box and white-box composition.

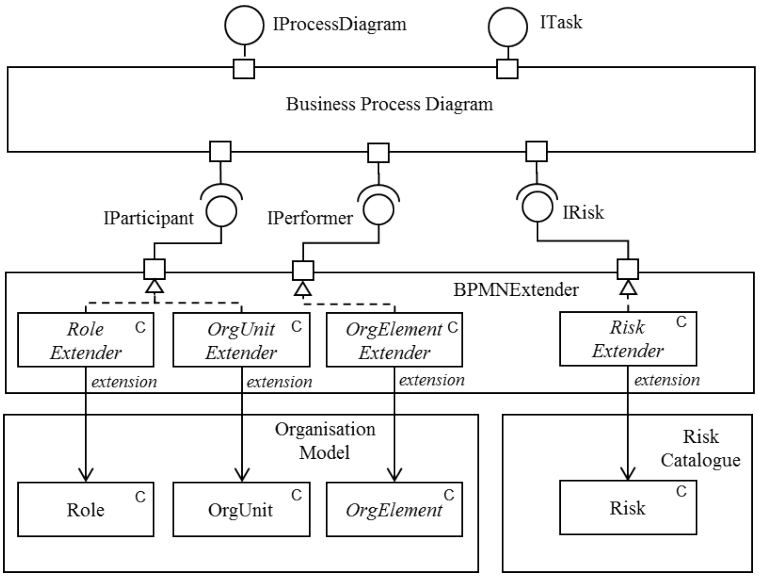


Figure 9.12: Achieving black-box independent composition of prefabricated metamodel fragments using dynamic invasive grey-box composition

- *Independent black-box composition.* In case of black-box fragments, the extender fragment may extend the realisation elements to declare the interface realisation of the fragment that contains the interface specification. In our case, the *Organisational Model* fragment and the *Risk Catalogue* fragment import interfaces from BPMN in order to declare interface realisation. We revisit the example from the black-box composition, in order to demonstrate how the dynamic invasive grey-box composition allows for, although semantically invasive (on the extendee side), independent composition of prefabricated fragments. Figure 9.12 illustrates the solution. We introduce the "glue logic" fragment *BPMNExtender*, which extends each of the realisation classes of the *Organisational Model* fragment and the *Risk Catalogue* fragment with interfaces from the BPD fragment. Hence, we extracted the glue logic out of the participating fragments to avoid their modification and to achieve independent fragment composition.

- *Independent white-box composition.* In case of the white-box composition, the extender fragment may extend elements of one fragment

with mixed in elements of another fragment. If we revisit the example of white-box composition introduced before, we adapt the solution by introducing the extender fragment *BPMNExtender* that overtakes the role of the composer fragment. Instead of the direct mixin inclusion by the class *GraphicalObject*, the extender class *GraphicalExtender* now includes the mixin *GraphicalMixin* and by virtue of extension, injects the structural features of the *GraphicalMixin* into the class *GraphicalObject* of the *BPMN* fragment. Hence, both fragments *BPMN* and *2DGraphical* remain unmodified achieving the independent composition. Figure 9.13 illustrates the solution.

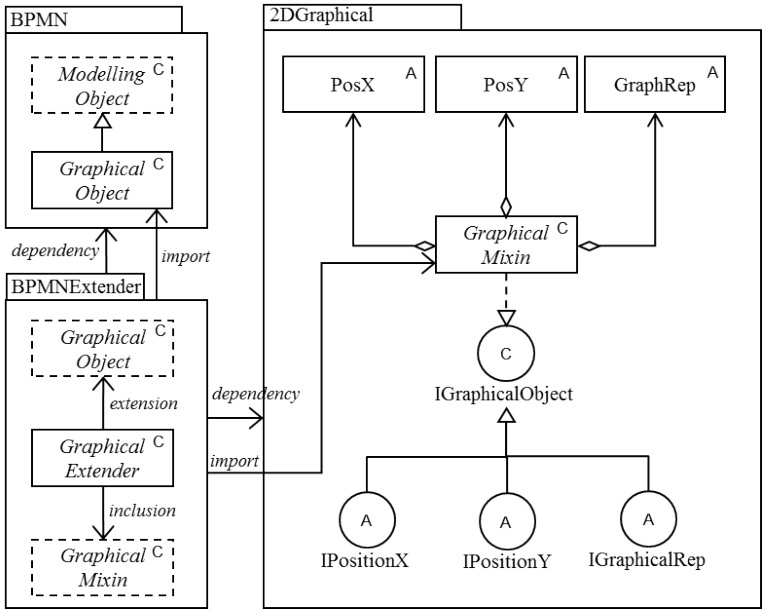


Figure 9.13: Achieving white-box independent composition of prefabricated metamodel fragments using dynamic invasive grey-box composition

9.2 Case Study in EA: Hybrid PDDSL

In this section we discuss the Hybrid DSL approach as introduced in Section 2.3, this time, in the context of modular metamodel engineering. To recall, Hybrid PDDSL is a hybrid modelling (domain-specific) language and approach for consistent modelling of network equipment. It introduces a domain-specific language for modelling and configuration of network devices, which is extended by the ontology language OWL2 [Motik et al., 2009b] to allow for and ease the semantic constraint definition of valid device configurations [Zivkovic et al., 2011, Miksa et al., 2013, Zivkovic et al., 2015].

We revisit the invasive metamodel integration solution proposed there, and explicate how the same integration problem may be solved non-invasively using modularisation and composition concepts and constructs of MMEL.

9.2.1 Revisiting the Integrated Metamodel Implementation

In Section 2.3 we provided a conceptual overview of the single metamodels of the Hybrid PDDSL, as well as of the corresponding metamodel integration solution, which has been invasive on the side of PDDSL. In the following, we take a closer look at the realisation of the integration approach and related issues.

Metamodel Composition based on Inheritance

The single metamodels and their integration relies on the instantiation of the constructs of the ADOxx Meta^2-Model (see Chapter 8). The notion of model type has been used to separate three major language blocks: *Device types*, *Devices* and *OWL2*. The model type OWL2 has been further split in two modes: *OWL2 Frames* and *OWL2 Descriptions*. On the other side, the notion of the *Interref relation* has been used to implement the ontological-instantiation relation between model types by defining interref-relationship *hasType* between *Artefact* and *ArtefactType*. Furthermore, the interref concept has been applied to achieve the metamodel integration between PDDSL and OWL2 by specifying the *generalisation relationships* between *pdArtefactType* and *owlClass*, as well as between *pdArtefact* and *pdIndividual*, respectively. Figure 9.14 illustrates the implementation of the metamodel integration.

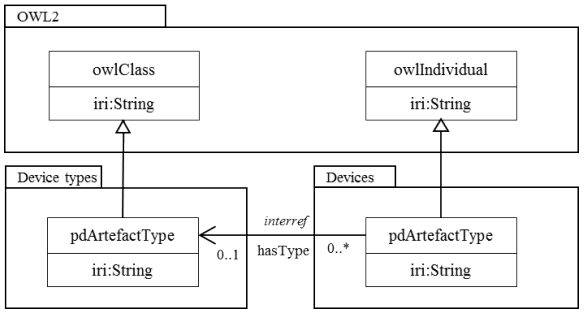


Figure 9.14: Hybrid PDDSL: Metamodel integration based on inheritance (simplified view of the integrated metamodel focusing on the integration points)

Issues with White-Box Composition based on Inheritance

Basically, by inheriting from OWL, the PDDSL classes, *pdArtefactType* and *pdArtefact* can be used in context of OWL2 whenever *owlClass* and *owlIndividual* are expected, respectively. Thus, the integration is non-invasive on the side of the OWL language. However, it is clearly invasive on the side of the PDDSL language, which might not always be a desirable implication if there are existing mechanisms that work on the modified metamodel. We can identify two major issues with inheritance-based solution as follows:

- *Root class precondition for extender class.* For pdArtefactType and pdArtefact classes to become subclasses of owlClass and owlIndividual, they must be root classes, i.e. do not have supertypes (restriction of single-inheritance of the underlying metamodelling language). This precondition can be rarely met, as metamodel element usually build class hierarchies within their own scope. In our case, this was possible, as we deliberately designed PDDSL for inheritance-based extension.

- *Undesired attribute inheritance.* If the root class precondition is met, the subclass will inherit all structural features, in our case, attributes from its superclass. While inheriting attributes is desirable in order to achieve the compatibility of the subtype with its supertype, in our case, it causes problems. In particular, the attribute *iri* of OWL classes is used as a unique identifier in OWL2, whereas the attribute *name* has the same role for PDDSL classes. By subclassing OWL classes with PDDSL classes, PDDSL classes inherit the *iri* attribute, although the attribute *name* already fulfills the role of the class identifier. Hence, it is not possible to state that an attribute from a superclass has the same role as the attribute in the subclass. As a consequence of this restriction, a modification of the PDDSL metamodel was required, in order to replace the *name* attribute with the attribute *iri*. This change, on the other side, implied the adaptation of metamodel-specific mechanisms specific to the PDDSL part of the language. In other words, the semantic compatibility was broken.

9.2.2 Modular Metamodel Definition using MMEL

In the following, we explicate how the issues of the pure white-box composition based on inheritance may be solved using the concepts and constructs of MMEL.

Modularisation of Metamodels

Clearly, modularisation of metamodels into fragments contributes to reuse. Packaging of OWL2 and PDDSL into black-box fragments with well defined interfaces hides the implementation details of each of the metamodels while

exposing only relevant parts. In addition, it also allows for reuse and recombination in other hybrid metamodel composition scenarios. Therefore, we first identify two reusable metamodel fragments for the OWL2 part, *owl2Frames* and *owl2Descriptions*. The fragment *owl2Frames* contributes with implementation concepts for basic modelling of class hierarchies and individuals. Therefore, we declare that it provides class interfaces *IOWLClass* and *IOWLIndividual*. In addition, we also define one provided attribute interface *IOWLIri*, which is aggregated as a member by both class interfaces. The *owl2Descriptions* fragment is a partial fragment that adds notions for defining OWL2 expressions (conjuctions, disjunctions, features such as object properties, etc.) on top of class and individual concepts. Thus, for *owl2Descriptions* we define required interfaces *IOWLCLass*, *IOWLIndividual* and *IOWLIri* to be kind of place holders for structural concepts needed to define OWL expressions. On the PDDSL side, we define two additional fragments *pdDeviceTypes*, and *pdDevices*. For the *pdDeviceTypes* fragment, we define one provided interface *IArtefactType*. Accordingly, the *pdDevices* fragment has one required interface *IArtefactType*. Figure 9.15 illustrates the introduced metamodel fragments as black-boxes, as a result of the interface-based modularisation process. Fragments are visualised as black-box components with interfaces based on the UML component diagram visual notation. Note that attribute interfaces are reusable elements, thus represented as first-order independent elements.

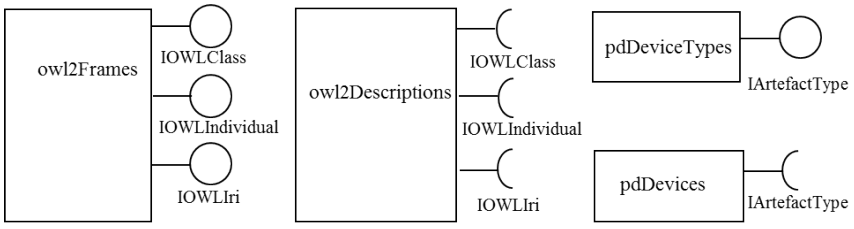


Figure 9.15: Reusable metamodel fragments as black-box components of the Hybrid PDDSL

Interface-based Black-box Composition

Having metamodel fragments with well-defined interfaces allows us to apply interface-based, black-box metamodel composition to create the hybrid language. From the modelling language perspective, our goal is to extend the PDDSL language in a way that OWL expressions are applicable for artefact types as they are for OWL classes. First, we compose the two fragments of the OWL language, by declaring that required interfaces from the *owlDescriptions* fragment are provided by their counterparts in *owlFrames*. We do this by using the *interfaceRealisation* operator, by declaring that the internal classes from owlFrames such as *owlCLass*, *owlIndividual*, *owlIri* realise

interfaces *IOWLClass*, *IOWLIndividual*, *IOWLIri*. This has an implication that whenever those interfaces are used within *owlDescriptions* the corresponding classes from *owlFrames* may be applied as compatible interface realisations. On the other side, we bind PDDSL fragments by declaring the realisation of the interface *IArtefactType* by the internal class *pdArtefactType* from *pdDeviceTypes*. Thus, whenever this interface is used within *pdDevices* fragment, the corresponding realisation class from *pdDeviceTypes* will be applied. Finally, as we aim at extending the artefact type for OWL class descriptions features, we declare that the *pdArtefactType* also realises the interface *IOWLClass*. By doing this, the *pdArtefactType* will become compatible realisation of the *IOWLClass*, whenever this interface is used in OWL2 descriptions, making it possible to define OWL2 expressions for device types. For the class interface realisation to be complete, attribute interface realisation need to be fulfilled, too. Hence, we use attributeInterfaceRealisation to declare that the existing attribute *pdName* implements the interface *IOWLIri*. Since both of the attribute elements are based on the same string type, the realisation is compatible and valid. Likewise, the internal class *pdArtefact* from *pdDevices* realises the *IOWLIndividual* interface. Obviously, we need to modify the PDDSL fragment, as we need to declare and provide realisations of the interfaces from OWL. We do this by introducing explicit dependencies from PDDSL to OWL, which is the prerequisite for interface element import. Figure 9.16 illustrates the metamodel fragment composition using the UML component diagram notation.

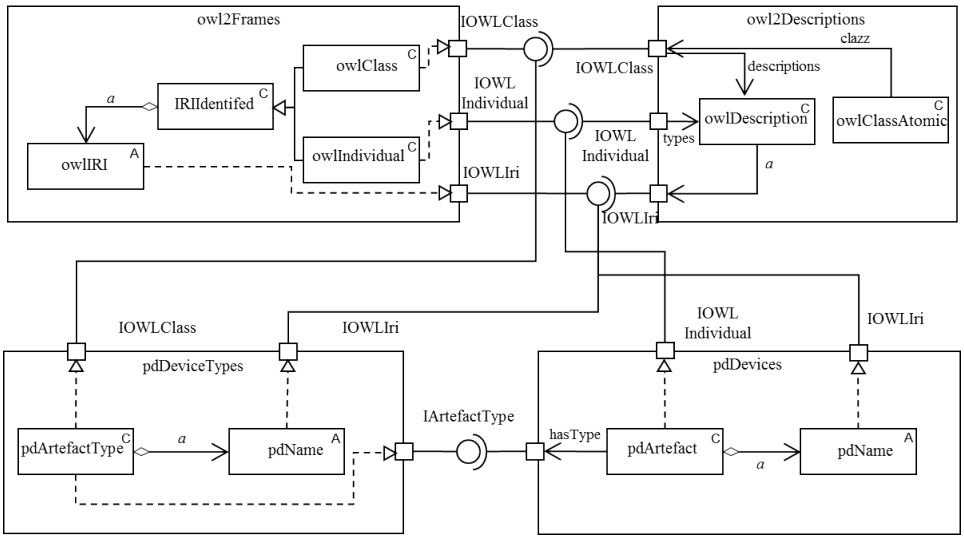


Figure 9.16: Black-box composition of metamodel fragments for Hybrid PDDSL

Using the black-box composition techniques, it becomes evident that we solved the issues that arise through the inheritance-based white-box com-

position discussed previously. Since a class may implement a number of interfaces, we resolve the issue of single inheritance, i.e. the s.c. root class precondition problem. Hence, the artefact type may internally define its own class hierarchy without interfering with other languages. Furthermore, since being able to define interface realisation on the attribute level, we have the flexibility to reuse existing implementation attributes for class interface realisation. Instead of inheriting the attribute, we established a kind of mapping between the required Iri attribute and the internal name attribute, without the need to semantically change the PDDSL language regarding the object identifier concept.

Extension-based Grey-Box Composition

In the following, we explicate how the introduced metamodel fragments may be combined using extension-based dynamic, invasive grey-box composition, in order to achieve independent fragment composition. This technique comes in handy in the cases where syntactic modification of existing fragments is not possible or not desirable. In our example before, we had to "open" the PDDSL fragments, to declare explicit dependencies, import interfaces and declare interface realisations. Using the grey-box composition techniques, we can achieve independent composition of prefabricated fragments by introducing a new composite extender fragment that will hold the glue logic of the two source fragments to be combined. The only prerequisite to apply grey-box composition is the public visibility of the elements to be extended. In our example, we will assume that this is case for the classes *pdArtefactType*, *pdArtefact* and *pdName*.

Figure 9.17 illustrates the solution. We introduce a new composite fragment that, on the one side, imports interfaces from OWL fragments, and on the other side injects interface realisation directives to the PDDSL fragments. In particular, the new composite fragment *hybPDDSL* introduces three extender elements that act as adapters between OWL2 and PDDSL elements. First, class extender *hybArtefactTypeExtender* extends the *pdArtefactType* at the extension point for interfaces by adding a new provided interface *IOWLClass*. This is done by declaring that the extender class realises the *IOWLClass*. By virtue of dynamic invasive extension, this interface is added to the list of supported interfaces of the *pdArtefactType*, without explicit need to change the *pdArtefactType* class definition. Since *IOWLClass* realisation requires the attribute interface *IOWLIri* to be accordingly realised by the implementing class, we introduce another extender for the attribute *pdName*, which injects the interface realisation of the interface *IOWLIri* to the name attribute. Finally, the third extender is defined in the same fashion, in order to inject the realisation of the interface *IOWLIndividual* at the PDDSL class *pdArtefact*.

As it becomes apparent, the end result of the composition is semantically

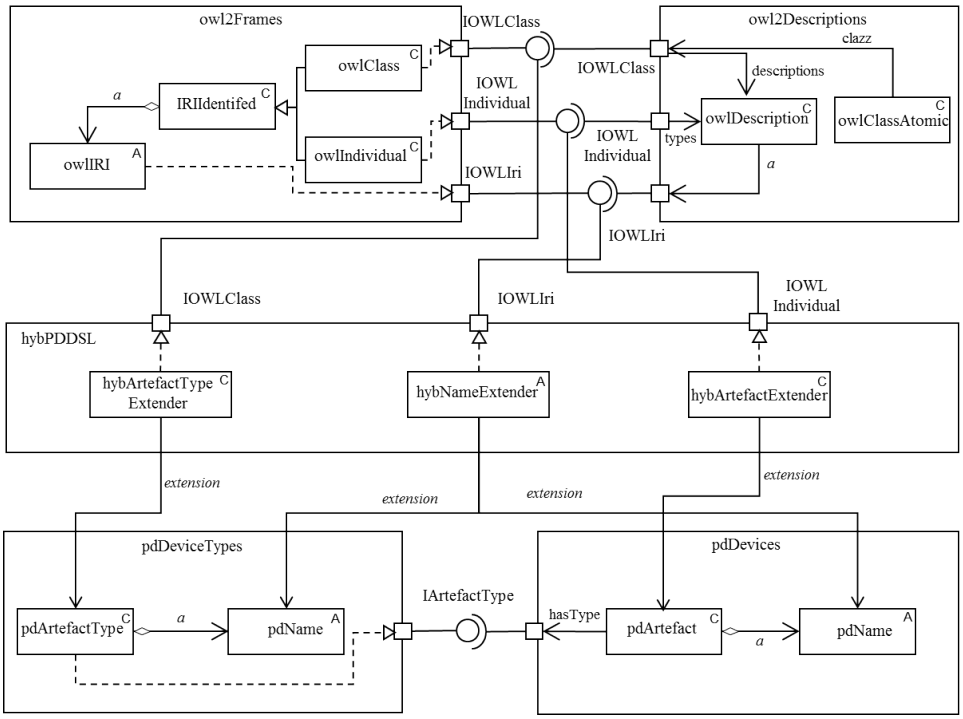


Figure 9.17: Extension-based grey-box composition of metamodel fragments for Hybrid PDDSL

equivalent to the black-box composition described previously. The difference lies in the syntactic immutability, i.e. in the independence of the participating metamodel fragments. While in black-box composition solution, the PDDSL fragments require modification to explicitly declare dependencies needed for interface realisation, the grey-box composition introduces one indirection level "in between". It defines a third fragment which holds the glue logic for the composition of the source fragments. Thus, the modifications that are needed on the PDDSL are packaged externally in the composition fragment and applied/injected to PDDSL fragments by virtue of grey-box invasive metamodel composition.

9.3 Chapter Summary

In this chapter, we evaluated the applicability of the MME approach in general, and of MMEL in particular. In particular, in Section 9.1 we demonstrated the usefulness of MME and its language MMEL on the example of the industry-relevant metamodel of the BPMS method for business process modelling. We have shown how such big and complex metamodel can be decomposed into modular, reusable metamodel fragments that can be flexibly combined to build different variants of the BPMS method, which

suit problem-, and project specific requirements. In particular, we demonstrated the usage of modularisation concepts such as metamodel fragment, metamodel element interface, as well as the usage of metamodel composition operators for black-box, grey-box and white-box metamodel composition. In Section 9.2 we revisited the case study of hybrid DSML in the field of EA and demonstrated how such integration may be done using MMEL. Here, we pointed out how deficiencies of a pure inheritance-based white-box composition may be overcome using black-box and grey-box operators such as interface realisation and dynamic extension. In particular, we demonstrated how metamodels can be encapsulated into reusable black-box fragments and combined based on explicit dependencies and interface-based composition. Furthermore, we also explicated how prefabricated metamodel fragments may be combined without modifications and direct dependencies on the composer side, by introducing the extender fragments based on invasive metamodel composition operators. Finally, as we stated already in [Živković and Karagiannis, 2015], it may be said that a modular approach for metamodel definition introduces a certain level of complexity in metamodel definition. This is mainly because additional (efficiency) constructs such as interfaces, fragments, extensions and alike are used apart from core (effectiveness) constructs when defining the modelling language. We acknowledge this argument against the modular approach only within the scope of metamodelling-in-the-small. However, the benefit of systematic modularisation becomes visible in larger metamodelling projects through economies of scale. By reusing and combining pre-fabricated fragments to build new metamodels significantly reduces development time and increases quality in terms of prepackaged solutions, all in the sense of metamodelling-in-the-large.

Part V

Summary

Chapter 10

Conclusion and Outlook

"The scientific man does not aim at an immediate result. He does not expect that his advanced ideas will be readily taken up. His work is like that of the planter – for the future. His duty is to lay the foundation for those who are to come, and point the way."

Nikola Tesla

In this chapter, we summarise the contributions of this work and provide an outlook for future work.

10.1 Conclusion

Acknowledging the rigorous significance of hybrid modelling methods, the inherent nature of modelling languages to evolve over time, the rapidly increasing number of languages to develop and maintain, and the inherent complexity of metamodel and language design, we concluded that existing metamodelling concepts do not provide sufficient means to adequately support such challenges of metamodel engineering. We used the metaphor of metamodelling-in-the-small to describe the current state of practice in metamodelling, and pointed out that concepts for metamodelling-in-the-large are not sufficiently supported.

By analysing the existing work, the following main deficiencies have been identified (reduced to the scope of this work):

- Within the approaches for modelling method integration, although high-level conceptual solutions for designing hybrid modelling methods in a modular way exist, sound formalisms for modular metamodelling on the technical level haven't been addressed (*Chapter II*).

- On the technical modelling language level, state-of-the-art metamodelling languages do not provide concepts for the modularisation of metamodels in terms of fragments with well-defined interfaces (*Chapter III, V*).
- Likewise, concepts for the flexible combination of metamodel fragments beyond standard operators on white-box packages are missing (*Chapter III, V*).
- Consequently, metamodelling platforms lack modular mechanisms for metamodel definition (*Chapter IV*).

Based on deficiencies found in the existing work in the field of metamodelling, the aim of the underlying work was to contribute to the state-of-the-art with a *modular approach to metamodel engineering by leveraging the concepts of metamodel modularisation and composition*. The work was an attempt to advance the state-of-the-art in metamodelling with conceptual, technical and qualitative contributions as follows.

1. A concept for modular metamodel engineering based on modularisation and composition concepts. The concept of reusable metamodel fragments with explicit interfaces and a set of composition operators that allow for mixin-based white-box, extension-based grey-box and interface-based black-box metamodel composition are the key contributions of the MME approach (*Chapter VI*).
2. A formal specification of a language for modular metamodel engineering as a modular extension to existing metamodelling language concepts (*Chapter VII*). The MMEL represents the cornerstone of the MME approach.
3. A technical specification of MMEL implementation in a metamodelling platform ADOxx (*Chapter VIII*), as an extension of the ADOxx metametamodel, ADOxx Meta^2-Model.
4. A qualitative evaluation of the applicability of MMEL based on the case study for modularising the metamodel of the BPMS modelling method (BP domain, primary focus on behavioural modelling concern) and the case study for developing a hybrid language PDDSL (EA domain, primary focus on structural modelling concern), both of them having relevant applications in research and industry (*Chapter IX*).

Besides main contributions, additional contributions beyond existing work, in the order of appereance in the chapters, are summarised as follows:

- Categorisation of modelling method hybrid mechanisms into configurable and adaptable (Section 2.1.4).

- Classification of modelling method mechanisms based on application purpose into analytical, computational, generative and manipulative (Section 2.1.4).
- Extension of the classification of hybrid modelling methods for a multi-dimensional category (Section 2.2.1).
- Case studies on hybrid modelling methods 1) Integration of BPMN and organisational modelling 2) multi-dimensional hybrid modelling method for interoperable, inter-organisational business processes 3) hybrid domain-specific modelling method for consistent physical devices management (Section 2.3).
- A detailed and extended definition of a modelling language anatomy considering the interface aspect (Section 3.2).
- A categorisation framework of metamodelling language capabilities (Section 3.4.2) and, based on it, a comparative analysis of metamodelling languages (Section 3.4.9).
- General classification of development environments (Section 4.1.3) and the generic architecture of language engineering environments, Section 4.2.1).
- A categorisation framework of metamodelling platform capabilities (Section 4.2.2) and, based on it, a comparative analysis of metamodelling platforms (Section 4.2.7).
- Classification of metamodel composition operators (Section 5.2).
- A classification framework (Section 5.1) and a comparative analysis of the metamodel modularisation and composition approaches (Section 5.3).

The introduced modular approach to metamodel engineering is specific to metamodel-based modelling language engineering. In particular, we evaluated the approach in the field of engineering of enterprise modelling methods. However, we believe that the modular approach may be applied to other modelling domains, too, as long as the modelling language definition is based on metamodels. The applicability of the approach for other language engineering formalisms (graph-based, grammar-based etc.) was not the focus of this work.

10.2 Outlook

In this section, we discuss the possible directions for future work related to the topic of modular metamodel and language engineering. Over the

years in research and practice in metamodelling and metamodelling tools development and in particular while working on the approach for modular metamodel engineering, a number of further open research questions came on to surface that would require a dedicated future research. In the following, some of the possible extensions to this work are sketched.

- *Modular Modelling Method Engineering.* As the focus of this work was on modular metamodel definition, we believe that the modular approach may further contribute to the overall flexibility in modelling method definition. A metamodel with explicit interfaces is a pivotal element, not only of a modelling language, but of a modelling method as well. This fact may be benefitial when building surrounding method components such as notations, mechanisms, algorithms and method procedures, that may refer to metamodel constructs via interfaces only. In doing so, the substitutability and overall reuse of metamodels and other method components may be significantly increased. We discussed how such an idea may be realised in ADOxx using the concept of MFBs in Section 8.4 of Chapter 8. However, how platform independent approaches and formalisms for modelling method engineering (e.g. Modelling Method DSL (MM-DSL) [Visic et al., 2015]) may be extended with modular specification of notations, mechanisms and algorithms and procedure models that interplay with modular metamodel definition towards modular modelling method engineering is left for future research.

- *Automation of Metamodelling-In-the-Large.* One of the main contributions of this work was to provide modularisation concepts for the encapsulation of self-contained, prefabricated metamodel fragments, and to define composition operators for flexible composition of those fragments. While we talk about fragments, the composition occurs on the interface, i.e. element level of fragments. The question is whether such composition may be automated by lifting the composition problem even one abstraction level higher to the level of metamodel fragments or even on the level of languages. A kind of domain-specific language that would contain operators such as *bind*, *extend* or *refine* which would automatically perform composition of fragments by compiling it to the composition operations such as those defined in this work would be desirable to further support the idea of the metamodelling-in-the-large.

- *Interdisciplinary Approaches for Modular Language Engineering.* Continuing the idea of the further advancements of metamodelling-in-the-large one can think of applying methodologies and approaches from other areas, such as software product lines engineering (SPL) or semantic technology, in the context of modular metamodel and language

definition. We already experimented with the idea of applying SPL feature-based modelling for customisation of model-driven software development environments [Wende et al., 2011]. In particular, in [Wende, 2012], feature-based approach for language family engineering has been successful applied. It would be interesting to investigate whether there are further fruitful integration points that could converge SPL and modular modelling language engineering. Likewise, in the research project MOST, the integration of metamodelling with semantic technologies has been discussed [Walter and Ebert, 2009, Aßmann et al., 2013]. In this context, we experimented with the idea of applying semantic reasoning for metamodel validation [Lekaditis, 2014]. As a starting point, the semantic technologies could be used in the context of modular metamodel engineering, for example, for the identification of compatible metamodel fragments, automatic matching of interfaces, etc.

- *Advanced Mechanisms for Hybrid Modelling.* The focus of our work has been primarily on the metamodel level and on corresponding composition mechanisms to allow for the definition of hybrid modelling languages. We introduced metamodelling capabilities of supporting character, i.e. for efficient metamodelling in the large. Some of such concepts, however, such as inheritance, interface realisation or interface subtyping have dual semantics. Not only that they allow for metamodel composition, they also lay foundation for the model-level mechanisms such as polymorphism, i.e. compatibility of objects on the model level. The question is however, whether there are additional mechanisms on the model level that would allow for advanced hybrid modelling scenarios currently not possible.

- *Collaborative Modular Metamodelling and Modelling in the Cloud.* It has been recognised that component orientation allows for distributed, independent development of components and their flexible composition as long as the implementation is streamlined based on agreed component interfaces. Having modelling languages and its fragments developed as components opens new opportunities for distributed metamodelling in the cloud. Collaborative metamodelling in distributed teams, management of metamodel component repositories, automatic discovery and integration of language components based on their compatible interfaces in the web as we know it from web services, provisioning of the adequate cloud-based metamodelling environments and supporting cloud services, and finally, application of the component orientation on the model level, are some of the directions where component-oriented metamodelling could advance in the upcoming years.

Bibliography

[ADOxx, 2015] ADOxx (2015). ADOxx Metamodelling Platform. http://www.adoxx.org.

[Agrawal, 2003] Agrawal, A. (2003). Graph Rewriting and Transformation (GReAT): A Solution for the Model Integrated Computing (MIC) Bottleneck. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 364–368. IEEE.

[Aho et al., 1986] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). Compilers, Principles, Techniques.

[Alderson, 1991] Alderson, A. (1991). Meta-CASE Technology. In Endres, A. and Weber, H., editors, *Software Development Environments and CASE Technology*, volume 509 of *Lecture Notes in Computer Science*, pages 81–91. Springer Berlin Heidelberg.

[Amelunxen et al., 2006] Amelunxen, C., Königs, A., Rötschke, T., and Schürr, A. (2006). MOFLON: A Standard-compliant Metamodeling Framework with Graph Transformations. In *Model Driven Architecture–Foundations and Applications*, pages 361–375. Springer.

[Ancona et al., 2000] Ancona, D., Lagorio, G., and Zucca, E. (2000). Jam- a Smooth Extension of Java with Mixins. In *ECOOP 2000 - Object-Oriented Programming*, pages 154–178. Springer.

[Aßmann, 2003] Aßmann, U. (2003). *Invasive Software Composition*. Springer.

[Aßmann et al., 2013] Aßmann, U., Zivkovic, S., Miksa, K., Siegemund, K., Bartho, A., Rahmani, T., Thomas, E., and Pan, J. (2013). Ontology-Guided Software Engineering in the MOST Workbench. In Pan, J. Z., Staab, S., Aßmann, U., Ebert, J., and Zhao, Y., editors, *Ontology-Driven Software Development*, pages 293–318. Springer Berlin Heidelberg.

[Atkinson and Kühne, 2003] Atkinson, C. and Kühne, T. (2003). Model-driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41.

[Bardohl et al., 1999] Bardohl, R., Minas, M., Taentzer, G., and Schürr, A. (1999). Application of Graph Transformation to Visual Languages. In Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G., editors, *Handbook of graph grammars and computing by graph transformation*, pages 105–180. World Scientific Publishing Co., Inc., River Edge, NJ, USA.

[Bartho et al., 2011] Bartho, A., Gröner, G., Rahmani, T., Zhao, Y., and Zivkovic, S. (2011). Guidance in Business Process Modelling. In *Service Engineering: European Research Results*, pages 201–231. Springer Vienna.

[Batini et al., 1986] Batini, C., Lenzerini, M., and Navathe, S. B. (1986). A Comparative Analysis of Methodologies for Database Schema Integration. *ACM computing surveys (CSUR)*, 18(4):323–364.

[Bayer and Kühn, 2013] Bayer, F. and Kühn, H. (2013). *Prozessmanagement für Experten: Impulse für aktuelle und wiederkehrende Themen*. Springer DE.

[Becker et al., 2009] Becker, J., Weiss, B., and Winkelmann, A. (2009). Developing a Business Process Modeling Language for the Banking Sector-A Design Science Approach. In *AMCIS*, page 709.

[Bézivin and Gerbé, 2001] Bézivin, J. and Gerbé, O. (2001). Towards a Precise Definition of the OMG/MDA Framework. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 273–280. IEEE.

[Bézivin et al., 2005] Bézivin, J., Hillairet, G., Jouault, F., Kurtev, I., and Piers, W. (2005). Bridging the MS/DSL Tools and the Eclipse Modeling Framework. In *Proceedings of the International Workshop on Software Factories at OOPSLA*, volume 5.

[Bézivin et al., 2004] Bézivin, J., Jouault, F., and Valduriez, P. (2004). On the Need for Megamodels.

[BOC, 2015] BOC (2015). Standard BPMS Method. http://www.boc-group.com/products/adonis/bpms-method-life-cycle/.

[Booch, 1991] Booch, G. (1991). *Object Oriented Design with Applications*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA.

[Bracha, 1992] Bracha, G. (1992). *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, The University of Utah.

[Bracha and Cook, 1990] Bracha, G. and Cook, W. (1990). Mixin-based Inheritance. In *ACM SIGPLAN Notices*, volume 25, pages 303–311. ACM.

[Bracha and Griswold, 1996] Bracha, G. and Griswold, D. (1996). Extending Smalltalk with Mixins. In *Workshop on Extending Smalltalk*.

[Bräuer and Lochmann, 2007] Bräuer, M. and Lochmann, H. (2007). Towards Semantic Integration of Multiple Domain-Specific Languages Using Ontological Foundations. In *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)*.

[Brinkkemper, 1996] Brinkkemper, S. (1996). Method engineering: engineering of information systems development methods and tools. *Information and Software Technology*, 38(4):275–280.

[Brinkkemper et al., 1999] Brinkkemper, S., Saeki, M., and Harmsen, F. (1999). Meta-Modelling based Assembly Techniques for Situational Method Engineering. *Information Systems*, 24(3):209–228.

[Bruckhaus et al., 1996] Bruckhaus, T., Madhavii, N., Janssen, I., and Henshaw, J. (1996). The Impact of Tools on Software Productivity. *Software, IEEE*, 13(5):29–38.

[Brunet et al., 2006] Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., and Sabetzadeh, M. (2006). A Manifesto for Model Merging. In *Proceedings of the 2006 international workshop on Global integrated model management*, pages 5–12. ACM.

[Calero et al., 2006] Calero, C., Ruiz, F., and Piattini, M. (2006). *Ontologies for Software Engineering and Software Technology*. Springer-Verlag.

[Charette, 1986] Charette, R. N. (1986). *Software Engineering Environments: Concepts and Technology*. Intertext Publications, Inc.,/McGraw-Hill, Inc.

[Chen, 1993] Chen, M. (1993). CASE Data Interchange Format (CDIF) Standards: Introduction and Evaluation. In *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on*, volume iii, pages 31–40 vol.3.

[Clark et al., 2014] Clark, T., Gonzalez-Perez, C., and Henderson-Sellers, B. (2014). A foundation for Multi-level Modelling. In *MULTI 2014–Multi-Level Modelling Workshop Proceedings*, page 43.

[Cook et al., 2007] Cook, S., Jones, G., Kent, S., and Wills, A. C. (2007). *Domain-specific Development with Visual Studio DSL Tools*. Pearson Education.

[COSO, 2013] COSO (2013). Enterprise Risk Management - Integrated Framework. ttp://www.coso.org/documents/COSO_ERM_ExecutiveSummary.pdf.

[Czarnecki and Helsen, 2006] Czarnecki, K. and Helsen, S. (2006). Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645.

[David et al., 2002] David, A., Möller, M. O., and Yi, W. (2002). Formal Verification of UML Statecharts with Real-time Extensions. In *Fundamental Approaches to Software Engineering*, pages 218–232. Springer.

[de Kinderen et al., 2012] de Kinderen, S., Gaaloul, K., and Proper, H. A. (2012). Bridging Value Modelling to ArchiMate via Transaction Modelling. *Software & Systems Modeling*, pages 1–15.

[de Lara and Guerra, 2013] de Lara, J. and Guerra, E. (2013). From Types to Type Requirements: Genericity for Model-Driven Engineering. *Software & Systems Modeling*, 12(3):453–474.

[De Lara and Vangheluwe, 2002] De Lara, J. and Vangheluwe, H. (2002). Using AToM as a Meta-CASE Tool. In *Proceedings of the 4th International Conference on Enterprise Information Systems (ICEIS)*, pages 642–649.

[de Lara Jaramillo et al., 2003] de Lara Jaramillo, J., Vangheluwe, H., and Alfonseca Moreno, M. (2003). Using Meta-modelling and Graph Grammars to Create Modelling Environments. *Electronic Notes in Theoretical Computer Science*, 72(3):36–50.

[Del Fabro et al., 2006] Del Fabro, M. D., Bézivin, J., and Valduriez, P. (2006). Weaving Models with the Eclipse AMW plugin. In *In Eclipse Modeling Symposium, Eclipse Summit Europe*.

[DeRemer and Kron, 1976] DeRemer, F. L. and Kron, H. H. (1976). Programming-in-the-Large Versus Programming-in-the-Small. In *Programmiersprachen*, pages 80–89. Springer.

[Dietrich et al., 2013] Dietrich, H., Breuker, D., Steinhorst, M., Delfmann, P., and Becker, J. (2013). Developing Graphical Model Editors for Meta-Modelling Tools–Requirements, Conceptualisation, and Implementation. *Enterprise Modelling and Information Systems Architectures*, 8(2):41–77.

[Dijkstra, 1959] Dijkstra, E. (1959). A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271.

[Dingel et al., 2008] Dingel, J., Diskin, Z., and Zito, A. (2008). Understanding and Improving UML Package Merge. *Software and Systems Modeling*, 7(4):443–467.

[Drey et al., 2010] Drey, Z., Faucher, C., Fleurey, F., Vincent, M., and Vojtisek, D. (2010). Kermeta Language - Reference Manual. http://www.kermeta.org/docs/fr.irisa.triskell.kermeta.documentation/build/pdf.fop/KerMeta-Manual/index.pdf.

[Ebert and Franzke, 1995] Ebert, J. and Franzke, A. (1995). A Declarative Approach to Graph Based Modeling. In *Graph-Theoretic Concepts in Computer Science*, pages 38–50. Springer.

[Ebert et al., 1996] Ebert, J., Winter, A., Dahm, P., Franzke, A., and Süttenbach, R. (1996). Graph Based Modeling and Implementation with EER/GRAL. In Thalheim, B., editor, *Conceptual Modeling ER '96*, volume 1157 of *Lecture Notes in Computer Science*, pages 163–178. Springer Berlin Heidelberg.

[Eclipse, 2013] Eclipse (2013). Eclipse Modelling Project. http://www.eclipse.org/modeling/.

[Eclipse, 2014a] Eclipse (2014a). Acceleo. http://www.eclipse.org/acceleo/.

[Eclipse, 2014b] Eclipse (2014b). Atlas Transformation Language. https://www.eclipse.org/atl/.

[Eclipse, 2014c] Eclipse (2014c). EMF Compare. http://www.eclipse.org/emf/compare/.

[Eclipse, 2014d] Eclipse (2014d). Graphical Modelling Project (GMP). http://www.eclipse.org/modeling/gmp/.

[Eclipse, 2014e] Eclipse (2014e). Model Execution Framework. https://www.eclipse.org/proposals/mxf/.

[Eclipse, 2014f] Eclipse (2014f). Textual Modelling Framework. http://www.eclipse.org/modeling/tmf/.

[Eclipse, 2014g] Eclipse (2014g). The CDO Model Repository. http://www.eclipse.org/cdo/.

[Emerson and Sztipanovits, 2006] Emerson, M. and Sztipanovits, J. (2006). Techniques for Metamodel Composition. In *OOPSLA 6th Workshop on Domain Specific Modeling*, pages 123–139.

[Feldman, 1979] Feldman, S. I. (1979). Make - A Program for Maintaining Computer Programs. *Software: Practice and experience*, 9(4):255–265.

[Fill and Karagiannis, 2013] Fill, H.-G. and Karagiannis, D. (2013). On the Conceptualisation of Modelling Methods using the ADOxx Meta Modelling Platform. *Enterprise Modelling and Information Systems Architectures-An International Journal*, 8(1).

[Floyd, 1967] Floyd, R. W. (1967). Assigning Meanings to Programs. *Mathematical aspects of computer science*, 19(19-32):1.

[Fowler, 2005] Fowler, M. (2005). Language Workbenches: The Killer-app for Domain Specific languages? Online Web Page.

[Frank, 2002] Frank, U. (2002). Multi-perspective Enterprise Modeling (MEMO) - Conceptual Framework and Modeling Languages. In *System Sciences, 2002. HICSS. Proceedings of the 35th Annual Hawaii International Conference on*, pages 1258–1267. IEEE.

[Fuentes-Fernández and Vallecillo-Moreno, 2004] Fuentes-Fernández, L. and Vallecillo-Moreno, A. (2004). An Introduction to UML Profiles. *UML and Model Engineering*, 2.

[Gérard et al., 2011] Gérard, S., Espinoza, H., Terrier, F., and Selic, B. (2011). 6 Modeling Languages for Real-Time and Embedded Systems. In *Model-Based Engineering of Embedded Real-Time Systems*, pages 129–154. Springer.

[Golin and Reiss, 1990] Golin, E. J. and Reiss, S. P. (1990). The Specification of Visual Language Syntax. *Journal of Visual Languages & Computing*, 1(2):141–157.

[Gonzalez-Perez and Henderson-Sellers, 2008] Gonzalez-Perez, C. and Henderson-Sellers, B. (2008). *Metamodelling for Software Engineering*. Wiley Publishing.

[Greenfield et al., 2004] Greenfield, J., Short, K., Cook, S., Kent, S., and Crupi, J. (2004). *Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools*. John Wiley & Sons.

[Gronback, 2009] Gronback, R. C. (2009). *Eclipse Modeling Project: A Domain-specific Language (DSL) Toolkit*. Pearson Education.

[Grundy and Venable, 1996] Grundy, J. C. and Venable, J. R. (1996). Towards an Integrated Environment for Method Engineering. In *Method Engineering*, pages 45–62. Springer.

[Happel and Seedorf, 2006] Happel, H. and Seedorf, S. (2006). Applications of Ontologies in Software Engineering. *Workshop on Semantic-Web Enabled Software Engineering (SWESE)*.

[Harel and Rumpe, 2000] Harel, D. and Rumpe, B. (2000). Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff.

[Harel and Rumpe, 2004] Harel, D. and Rumpe, B. (2004). Meaningful Modeling: What's the Semantics of. *Computer*, 37(10):64–72.

[Harmsen, 1997] Harmsen, A. F. (1997). Situational Method Engineering. *Moret Ernst & Young Management Consultants*.

[Harrison and Ossher, 1993] Harrison, W. and Ossher, H. (1993). *Subject-oriented Programming: A Critique of Pure Objects*, volume 28. ACM.

[Heidenreich et al., 2013] Heidenreich, F., Johannes, J., Karol, S., Seifert, M., and Wende, C. (2013). Model-Based Language Engineering with EMFText. In *Generative and Transformational Techniques in Software Engineering IV*, pages 322–345. Springer.

[Helm and Marriott, 1991] Helm, R. and Marriott, K. (1991). A Declarative Specification and Semantics for Visual Languages. *Journal of Visual Languages & Computing*, 2(4):311–331.

[Henderson-Sellers and Gonzalez-Perez, 2005] Henderson-Sellers, B. and Gonzalez-Perez, C. (2005). A Comparison of Four Process Metamodels and the Creation of a New Generic Standard. *Information and Software Technology*, 47(1):49–65.

[Henderson-Sellers et al., 2008] Henderson-Sellers, B., Gonzalez-Perez, C., and Ralyté, J. (2008). Comparison of Method Chunks and Method Fragments for Situational Method Engineering. In *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*, pages 479–488. IEEE.

[Henderson-Sellers and Ralyté, 2010] Henderson-Sellers, B. and Ralyté, J. (2010). Situational method engineering: State-of-the-art review. *J. UCS*, 16(3):424–478.

[Hessellund et al., 2007] Hessellund, A., Czarnecki, K., and Wąsowski, A. (2007). Guided Development with Multiple Domain-specific Languages. In *Model Driven Engineering Languages and Systems*, pages 46–60. Springer.

[Hopcroft, 1971] Hopcroft, J. E. (1971). An N log N Algorithm for Minimizing States in a Finite Automaton. Technical report, Stanford University, Stanford, CA, USA.

[Horridge and Patel-Schneider, 2009] Horridge, M. and Patel-Schneider, P. F. (2009). OWL 2 Web Ontology Language Manchester Syntax . http://www.w3.org/TR/owl2-manchester-syntax/.

[Hudak, 1998] Hudak, P. (1998). Modular Domain Specific Languages and Tools. In *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press.

[IFIP-IPAC Task Force, 1999] IFIP-IPAC Task Force (1999). GERAM: Generalised Enterprise Reference Architecture and Methodology, Version

1.6.3. http://www.ict.griffith.edu.au/~bernus/taskforce/geram/versions/geram1-6-3/GERAMv1.6.3.pdf.

[Isazadeh and Lamb, 1997] Isazadeh, H. and Lamb, D. A. (1997). A Comparative Review of MetaCASE Tools. In *Systems Development Methods for the Next Century*, pages 297–311. Springer.

[Jackson, 1990] Jackson, M. (1990). Some Complexities in Computer-based Systems and Their Implications for System Development. In *CompEuro'90. Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering*, pages 344–351. IEEE.

[Jacobson et al., 1999] Jacobson, I., Booch, G., and Rumbaugh, J. (1999). *The Unified Software Development Process (Paperback)*. Addison-Wesley Object Technology. Pearson Education, Limited.

[JetBrains, 2013] JetBrains (2013). JetBrains, Meta Programming System, Documentation. http://www.jetbrains.com/mps/documentation/index.html.

[Johnson, 1975] Johnson, S. C. (1975). *Yacc: Yet Another Compiler-Compiler*, volume 32. Bell Laboratories Murray Hill, NJ.

[Jouault and Bézivin, 2006] Jouault, F. and Bézivin, J. (2006). KM3: a DSL for Metamodel Specification. In *Formal Methods for Open Object-Based Distributed Systems*, pages 171–185. Springer.

[Jouault et al., 2010] Jouault, F., Vanhooff, B., Bruneliere, H., Doux, G., Berbers, Y., and Bezivin, J. (2010). Inter-DSL Coordination Support by Combining Megamodeling and Model Weaving. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2011–2018. ACM.

[Junginger et al., 2000] Junginger, S., Kühn, H., Strobl, R., and Karagiannis, D. (2000). Ein Geschäftsprozessmanagement-Werkzeug der Nächsten Generation - ADONIS: Konzeption und Anwendungen. *Wirtschaftsinformatik*, 42(5):392–401.

[Karagiannis, 1995] Karagiannis, D. (1995). BPMS: Business Process Management Systems. *ACM SIGOIS Bulletin*, 16(1):10–13.

[Karagiannis, 2015] Karagiannis, D. (2015). Agile Modeling Method Engineering. In *Proceedings of the 19th Panhellenic Conference on Informatics*, pages 5–10. ACM.

[Karagiannis and Kühn, 2002] Karagiannis, D. and Kühn, H. (2002). Metamodelling Platforms. Invited Paper. In *Proceedings of the Third International Conference EC-Web 2002 - Dexa 2002*. Springer-Verlag, Berlin, Heidelberg.

[Karsai et al., 2003] Karsai, G., Agrawal, A., Shi, F., and Sprinkle, J. (2003). On the Use of Graph Transformation in the Formal Specification of Model Interpreters. *J. UCS*, 9(11):1296–1321.

[Karsai et al., 2004] Karsai, G., Maroti, M., Lédeczi, Á., Gray, J., and Sztipanovits, J. (2004). Composition and Cloning in Modeling and Metamodeling. *Control Systems Technology, IEEE Transactions on*, 12(2):263–278.

[Keller et al., 1992] Keller, G., Scheer, A.-W., and Nüttgens, M. (1992). *Semantische Prozeßmodellierung auf der Grundlage "Ereignisgesteuerter Prozeßketten (EPK)"*. Inst. für Wirtschaftsinformatik.

[Kelly et al., 1996] Kelly, S., Lyytinen, K., and Rossi, M. (1996). Metaedit+ A Fully Configurable Multi-user and Multi-tool CASE and CAME Environment. In *Advanced Information Systems Engineering*, pages 1–21. Springer.

[Kelly and Tolvanen, 2008] Kelly, S. and Tolvanen, J.-P. (2008). *Domain-specific Modeling: Enabling Full Code Generation*. Wiley.

[Kent, 2002] Kent, S. (2002). Model Driven Engineering. In *Integrated Formal Methods*, pages 286–298. Springer.

[Klein, 2001] Klein, M. (2001). Combining and Relating Ontologies: An Analysis of Problems and Solutions. In *IJCAI-2001 Workshop on ontologies and information sharing*, pages 53–62.

[Kleppe, 2007] Kleppe, A. (2007). Towards the Generation of a Text-based IDE From a Language Metamodel. In *Model Driven Architecture-Foundations and Applications*, pages 114–129. Springer.

[Kleppe, 2009] Kleppe, A. (2009). *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional.

[Kleppe et al., 2003] Kleppe, A., Warmer, J., and Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley.

[Kolovos et al., 2006] Kolovos, D., Paige, R., and Polack, F. (2006). Merging Models with the Epsilon Merging Language (EML). In Nierstrasz, O., Whittle, J., Harel, D., and Reggio, G., editors, *Model Driven Engineering Languages and Systems*, volume 4199, chapter 16, pages 215–229. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Kolovos et al., 2010] Kolovos, D. S., Rose, L. M., Abid, S. B., Paige, R. F., Polack, F. A., and Botterweck, G. (2010). Taming EMF and GMF Using Model Transformation. In *Model Driven Engineering Languages and Systems*, pages 211–225. Springer.

[Kruchten, 2004] Kruchten, P. (2004). *The Rational Unified Process: An Introduction*. Addison-Wesley Professional.

[Kühn, 2004] Kühn, H. (2004). *Method Integration in Business Engineering (In German)*. PhD thesis, Faculty of Computer Science, University of Vienna, Austria.

[Kühn, 2010] Kühn, H. (2010). The ADOxx® Metamodelling Platform. In *Workshop on Methods as Plug-Ins for Meta-Modelling, Klagenfurt, Austria*.

[Kühn et al., 2003] Kühn, H., Bayer, F., Junginger, S., and Karagiannis, D. (2003). Enterprise model integration. In *EC-Web*, pages 379–392.

[Kühn et al., 2001] Kühn, H., Junginger, S., Bayer, F., and Petzmann, A. (2001). Managing Complexity in E-Business. In *Proceedings of the 8th European Concurrent Engineering Conference*, pages 6–11.

[Kühn et al., 2011] Kühn, H., Murzek, M., Specht, G., and Zivkovic, S. (2011). Model-Driven Development of Interoperable, Inter-Organisational Business Processes. In Charalabidis, Y., editor, *Interoperability in Digital Public Services and Administration: Bridging E-Government and E-Business*, pages 119–143. Hershey, PA, USA.

[Langer et al., 2012] Langer, P., Wieland, K., Wimmer, M., Cabot, J., et al. (2012). EMF Profiles: A Lightweight Extension Approach for EMF Models. *Journal of Object Technology*, 11(1):1–29.

[Lédeczi et al., 2001] Lédeczi, Á., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., and Karsai, G. (2001). Composing Domain-specific Design Environments. *Computer*, 34(11):44–51.

[Ledeczi et al., 2001a] Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., and Volgyesi, P. (2001a). The Generic Modeling Environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, volume 17.

[Ledeczi et al., 2001b] Ledeczi, A., Nordstrom, G., Karsai, G., Volgyesi, P., and Maroti, M. (2001b). On Metamodel Composition. In *Control Applications, 2001. (CCA '01). Proceedings of the 2001 IEEE International Conference on Control Applications*, pages 756–760.

[Lekaditis, 2014] Lekaditis, C. (2014). *Validation of ADOxx Metamodels based on Semantic Technologies.* Master thesis, Faculty of Computer Science, University of Vienna, Austria.

[Lemcke et al., 2010] Lemcke, J., Rahmani, T., and Friesen, A. (2010). Semantic Business Process Engineering. In *Reasoning Web. Semantic Technologies for Software Engineering*, pages 161–181. Springer.

[Lochmann and Hessellund, 2009] Lochmann, H. and Hessellund, A. (2009). An Integrated View on Modeling with Multiple Domain-specific Languages. In *Proceedings of the IASTED International Conference Software Engineering (SE 2009)*, pages 1–10.

[Marriott et al., 1998] Marriott, K., Meyer, B., and Wittenburg, K. B. (1998). A Survey of Visual Language Specification and Recognition. In *Visual language theory*, pages 5–85. Springer.

[Martin, 1994] Martin, C. (1994). MetaCASE: Dream or Reality. In *Electro/94 International. Conference Proceedings. Combined Volumes.*, pages 195–199.

[McDavid, 2004] McDavid, D. (2004). The Business-IT Gap: A Key Challenge. http://www.almaden.ibm.com/coevolution/pdf/mcdavid.pdf.

[MetaCase, 2011] MetaCase (2011). MetaEdit+ User's Guide, Version 4.5. http://www.metacase.com/support/45/manuals/meplus/Mp.html.

[Microsoft, 2013] Microsoft (2013). Common Language Runtime (CLR). http://msdn.microsoft.com/en-us/library/8bs2ecf4%28v=vs.110%29.aspx.

[Microsoft, 2014a] Microsoft (2014a). .NET Framework. http://msdn.microsoft.com/en-us/library/w0x726c2%28v=vs.110%29.aspx.

[Microsoft, 2014b] Microsoft (2014b). Visual Studio 2013. http://msdn.microsoft.com/en-us/library/ff361664%28v=vs.110%29.aspx.

[Miksa et al., 2013] Miksa, K., Sabina, P., Friesen, A., Rahmani, T., Lemcke, J., Wende, C., Zivkovic, S., Aßmann, U., and Bartho, A. (2013). Case Studies for Marrying Ontology and Software Technologies. In Pan, J. Z., Staab, S., Aßmann, U., Ebert, J., and Zhao, Y., editors, *Ontology-Driven Software Development*, pages 69–94. Springer Berlin Heidelberg.

[Miksa et al., 2010] Miksa, K., Sabina, P., and Kasztelnik, M. (2010). Combining Ontologies with Domain Specific Languages: A Case Study from Network Configuration Software. In Aßmann, U., Bartho, A., and Wende, C., editors, *Reasoning Web. Semantic Technologies for Software Engineering*, volume 6325 of *Lecture Notes in Computer Science*, pages 99–118. Springer Berlin Heidelberg.

[Mirbel and Ralyté, 2005] Mirbel, I. and Ralyté, J. (2005). Situational Method Engineering: Combining Assembly-based and Roadmap-driven Approaches. *Requirements Engineering*, 11:58–78.

[Moody, 2009] Moody, D. (2009). The "physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *Software Engineering, IEEE Transactions on*, 35(6):756–779.

[Moon, 1986] Moon, D. A. (1986). Object-oriented programming with Flavors. In *ACM Sigplan Notices*, volume 21, pages 1–8. ACM.

[Morin et al., 2009] Morin, B., Perrouin, G., Lahire, P., Barais, O., Vanwormhoudt, G., and Jézéquel, J.-M. (2009). Weaving Variability into Domain Metamodels. In *Model driven engineering languages and systems*, pages 690–705. Springer.

[Mosses, 1996] Mosses, P. D. (1996). *Theory and Practice of Action Semantics*. Springer.

[Motik et al., 2009a] Motik, B., Patel-Schneider, P. F., and Parcia, B. (2009a). OWL 2 Web Ontology Language Direct Semantics . http://www.w3.org/TR/2009/REC-owl2-direct-semantics-20091027/.

[Motik et al., 2009b] Motik, B., Patel-Schneider, P. F., and Parcia, B. (2009b). OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax . http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/.

[MSDN, 2012] MSDN (2012). Visualization and Modeling SDK - Domain-Specific Languages. http://msdn.microsoft.com/en-us/library/bb126259.aspx.

[MSDN, 2015] MSDN (2015). Extension Methods (C# Programming Guide). https://msdn.microsoft.com/en-us//library/bb383977.aspx.

[Mukerji and Miller, 2003] Mukerji, J. and Miller, J. (2003). MDA Guide Version 1.0.1. http://www.omg.org/cgi-bin/doc?omg/03-06-01.

[Murzek et al., 2013] Murzek, M., Rausch, T., and Kühn, H. (2013). BPMN als Bestandteil der BPMS-Modellierungsmethode. In Bayer, F. and Kühn, H., editors, *Prozessmanagement für Experten*, pages 93–113. Springer Berlin Heidelberg.

[Noy, 2004] Noy, N. F. (2004). Semantic Integration: A Survey of Ontology-based Approaches. *ACM Sigmod Record*, 33(4):65–70.

[Odersky et al., 2004] Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and

Zenger, M. (2004). An overview of the Scala programming language. Technical report, EPFL.

[OMG, 2008] OMG (2008). Software and Systems Process Engineering Specification (SPEM) Version 2.0 . http://www.omg.org/spec/SPEM/2.0/PDF/.

[OMG, 2011a] OMG (2011a). UML 2.4.1 Infrastructure Specification. http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/.

[OMG, 2011b] OMG (2011b). UML 2.4.1 Superstructure Specification. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF.

[OMG, 2012] OMG (2012). Object Constraint Language (OCL) Version 2.3.1. http://www.omg.org/spec/OCL/2.3.1/PDF.

[OMG, 2013] OMG (2013). Business Process Model and Notation (BPMN) Version 2.0.2. http://www.omg.org/spec/BPMN/2.0.2/PDF.

[OMG, 2014] OMG (2014). Meta Object Facility (MOF) Version 2.4.2. http://www.omg.org/spec/MOF/2.4.2/.

[OMI, 2015] OMI (2015). Open Model Initiative Laboratory. http://www.omilab.at.

[Oracle, 2013] Oracle (2013). Java Virtual Machine Specification. http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-1.html#jvms-1.2.

[Ouksel and Sheth, 1999] Ouksel, A. M. and Sheth, A. (1999). Semantic Interoperability in Global Information Systems. *ACM Sigmod Record*, 28(1):5–12.

[Pan et al., 2013] Pan, J. Z., Staab, S., Aßmann, U., Ebert, J., and Zhao, Y., editors (2013). *Ontology-Driven Software Development*. Springer, Berlin.

[Parker, 1992] Parker, B. (1992). Introducing ANSI-X3.138-1988: A Standard for Information Resource Dictionary System (IRDS). In *Assessment of Quality Software Development Tools, 1992., Proceedings of the Second Symposium on*, pages 90–99.

[Pedro et al., 2008] Pedro, L., Amaral, V., and Buchs, D. (2008). Foundations for a Domain Specific Modeling Language Prototyping Environment: A Compositional Approach. In *Proc. 8th OOPSLA ACM-SIGPLAN Workshop on Domain-Specific Modeling (DSM)*. University of Jyväskylän.

[Plotkin, 1981] Plotkin, G. D. (1981). *A Structural Approach to Operational Semantics*. DAIMI Aarhus University, Denmark.

[Pottinger and Bernstein, 2003] Pottinger, R. A. and Bernstein, P. A. (2003). Merging Models Based on Given Correspondences. In *VLDB '2003: Proceedings of the 29th international conference on very large data bases*, pages 862–873. VLDB Endowment.

[Prackwieser et al., 2013] Prackwieser, C., Buchmann, R., Grossmann, W., and Karagiannis, D. (2013). Towards a Generic Hybrid Simulation Algorithm Based on a Semantic Mapping and Rule Evaluation Approach. In *Knowledge Science, Engineering and Management*, pages 147–160. Springer.

[Ralyté, 1999] Ralyté, J. (1999). Reusing Scenario Based Approaches in Requirement Engineering Methods: CREWS Method Base. In *Proceedings of the 10th International Workshop on Database & Expert Systems Applications*, DEXA '99, pages 305–, Washington, DC, USA. IEEE Computer Society.

[Ralyté, 2004] Ralyté, J. (2004). Towards Situational Methods for Information Systems Development: Engineering Reusable Method Chunks. In *Procs. 13th Int. Conf. on Information Systems Development. Advances in Theory, Practice and Education*, pages 271–282.

[Ralyté et al., 2006] Ralyté, J., Backlund, P., Kühn, H., and Jeusfeld, M. A. (2006). Method Chunks for Interoperability. In *Conceptual Modeling-ER 2006*, pages 339–353. Springer.

[Ralyté et al., 2003] Ralyté, J., Deneckère, R., and Rolland, C. (2003). Towards a Generic Model for Situational Method Engineering. In *Advanced Information Systems Engineering*, pages 95–110. Springer.

[Ralyté and Rolland, 2001] Ralyté, J. and Rolland, C. (2001). An assembly process model for method engineering. In *CAiSE*, pages 267–283.

[Ralyté et al., 2004] Ralyté, J., Rolland, C., and Deneckère, R. (2004). Towards a meta-tool for change-centric method engineering: A typology of generic operators. In *CAiSE*, pages 202–218.

[Rausch et al., 2011] Rausch, T., Kuehn, H., Murzek, M., and Brennan, T. (2011). Making BPMN 2.0 Fit for Full Business Use. *Bpmn 2.0 Handbook Second Edition*, page 189.

[Rekers, 1995] Rekers, J. (1995). On the Use of Graph Grammars for Defining the Syntax of Graphical Languages. In *Proc. Colloquium on Graph Transformation and its Application in Computer Science. Technical Report B-19, Universitat de les Illes Balears*. Citeseer.

[Ren et al., 2009] Ren, Y., Gröner, G., Lemcke, J., Rahmani, T., Friesen, A., Zhao, Y., Pan, J. Z., and Staab, S. (2009). Validating Process Refinement with Ontologies. *Description Logics*, 477.

[Ren et al., 2013] Ren, Y., Gröner, G., Lemcke, J., Rahmani, T., Friesen, A., Zhao, Y., Pan, J. Z., and Staab, S. (2013). Process Refinement Validation and Explanation with Ontology Reasoning. In *Service-Oriented Computing*, pages 515–523. Springer.

[Rolland et al., 1999] Rolland, C., Prakash, N., and Benjamen, A. (1999). A Multi-Model View of Process Modelling. *Requirements Engineering*, 4(4):169–187.

[Scheer, 1992] Scheer, A.-W. (1992). Architektur integrierter Informationssysteme. *Berlin, Heidelberg*.

[Schroth et al., 2007] Schroth, C., Pemptroad, G., and Janner, T. (2007). CCTS-based Business Information Modelling for Increasing Cross-Organizational Interoperability. In *Enterprise Interoperability II*, pages 467–478. Springer.

[Schurr et al., 1995] Schurr, A., Winter, A., and Zundorf, A. (1995). Visual Programming with Graph Rewriting Systems. In *Visual Languages, Proceedings., 11th IEEE International Symposium on*, pages 326–333. IEEE.

[Schwarz et al., 2010] Schwarz, H., Ebert, J., Lemcke, J., Rahmani, T., and Zivkovic, S. (2010). Using Expressive Traceability Relationships for Ensuring Consistent Process Model Refinement. In *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, pages 183–192. IEEE.

[Scott and Strachey, 1971] Scott, D. and Strachey, C. (1971). *Toward a Mathematical Semantics for Computer Languages*, volume 1. Oxford University Computing Laboratory, Programming Research Group.

[Selic, 2007] Selic, B. (2007). A Systematic Approach to Domain-specific Language Design using UML. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium on*, pages 2–9. IEEE.

[Selic, 2011] Selic, B. (2011). The Theory and Practice of Modeling Language Design for Model-Based Software Engineering - A Personal Perspective. In *Generative and Transformational Techniques in Software Engineering III*, pages 290–321. Springer.

[Shen et al., 2004] Shen, H., Wall, B., Zaremba, M., Chen, Y., and Browne, J. (2004). Integration of Business Modelling Methods for Enterprise Infor-

mation System Analysis and User Requirements Gathering. *Computers in Industry*, 54(3):307–323.

[Smaragdakis and Batory, 2001] Smaragdakis, Y. and Batory, D. (2001). Mixin-based Programming in C++. In *Generative and Component-based Software Engineering*, pages 164–178. Springer.

[Staab et al., 2010] Staab, S., Walter, T., GrÃ¶ner, G., and Parreiras, F. (2010). Model Driven Engineering with Ontology Technologies. In Aßmann, U., Bartho, A., and Wende, C., editors, *Reasoning Web. Semantic Technologies for Software Engineering*, volume 6325 of *Lecture Notes in Computer Science*, pages 62–98. Springer Berlin Heidelberg.

[Steinberg et al., 2008] Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: Eclipse Modeling Framework*. Pearson Education.

[Szyperski, 2002] Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

[Tan and Liu, 2012] Tan, S. L. and Liu, L. (2012). Performance Analysis of Reusable Components with Hybrid Modelling of Strategies and Processes: A Real World Case Study. In *Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual*, pages 302–309. IEEE.

[The Open Group, 2012] The Open Group (2012). TOGAF, Enterprise Edition, Version 9.1. http://pubs.opengroup.org/architecture/togaf9-doc/arch/index.html.

[The Open Group, 2013] The Open Group (2013). ArchiMate, Version 2.0. http://www.opengroup.org/subjectareas/enterprise/archimate.

[Tolvanen, 1998] Tolvanen, J.-P. (1998). *Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence*. University of Jyväskylä.

[Tolvanen and Kelly, 2010] Tolvanen, J.-P. and Kelly, S. (2010). Integrating Models with Domain-specific Modeling Languages. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*, page 10. ACM.

[Tolvanen et al., 2007] Tolvanen, J.-P., Pohjonen, R., and Kelly, S. (2007). Advanced Tooling for Domain-specific Modeling: MetaEdit+. In *Sprinkle, J., Gray, J., Rossi, M., Tolvanen, JP (eds.) The 7th OOPSLA Workshop on Domain-Specific Modeling, Finland*.

[Vallecillo, 2010] Vallecillo, A. (2010). On the Combination of Domain Specific Modeling Languages. In Kühne, T., Selic, B., Gervais, M.-P., and Terrier, F., editors, *Proceedings of European Conference on Modelling Foundations and Applications, 2010. (ECMFA 2010)*, volume 6138 of *Lecture Notes in Computer Science*, pages 305–320. Springer.

[Venable, 1994] Venable, J. R. (1994). *CoCoA: A Conceptual Data Modelling Approach for Complex Problem Domains*. PhD thesis, State University of New York at Binghamton, Watson School of Engineering and Applied Science.

[Vernadat, 2006] Vernadat, F. (2006). The CIMOSA Languages. In Bernus, P., Mertins, K., and Schmidt, G., editors, *Handbook on Architectures of Information Systems*, International Handbooks on Information Systems, pages 251–272. Springer Berlin Heidelberg.

[Visic et al., 2015] Visic, N., Fill, H.-G., Buchmann, R., and Karagiannis, D. (2015). A Domain-specific Language for Modeling Method Definition: From Requirements to Grammar. In *Research Challenges in Information Science (RCIS), 2015 IEEE 9th International Conference on*, pages 286–297.

[Visser, 2008] Visser, E. (2008). WebDSL: A Case Study in Domain-Specific Language Engineering. *Generative and Transformational Techniques in Software Engineering II*, 5235:291–373.

[Voelter, 2013] Voelter, M. (2013). Language and IDE Modularization and Composition with MPS. In *Generative and transformational techniques in software engineering IV*, pages 383–430. Springer.

[Voelter and Solomatov, 2010] Voelter, M. and Solomatov, K. (2010). Language Modularization and Composition with Projectional Language Workbenches Illustrated with MPS. *Software Language Engineering, SLE*.

[Wachsmuth, 2007] Wachsmuth, G. (2007). Metamodel Adaptation and Model Co-adaptation. In *ECOOP 2007–Object-Oriented Programming*, pages 600–624. Springer.

[Walter and Ebert, 2009] Walter, T. and Ebert, J. (2009). Combining DSLs and Ontologies Using Metamodel Integration. In *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*, pages 148–169, Berlin, Heidelberg. Springer-Verlag.

[Walter and Ebert, 2011] Walter, T. and Ebert, J. (2011). Foundations of Graph-based Modeling Languages. Technical report, Technical report, University of Koblenz-Landau, Institute for Software Technology.

[Warmer and Kleppe, 2006] Warmer, J. B. and Kleppe, A. G. (2006). Building a Flexible Software Factory Using Partial Domain Specific Models. In *Sixth OOPSLA Workshop on Domain-Specific Modeling (DSM'06), Portland, Oregon, USA*, pages 15–22. University of Jyvaskyla.

[Weisemöller et al., 2011] Weisemöller, I., Klar, F., and Schürr, A. (2011). Development of Tool Extensions with MOFLON. In *Model-Based Engineering of Embedded Real-Time Systems*, pages 337–343. Springer.

[Weisemöller and Schürr, 2008] Weisemöller, I. and Schürr, A. (2008). Formal Definition of MOF 2.0 Metamodel Components and Composition. In *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 386–400, Berlin, Heidelberg. Springer-Verlag.

[Wende, 2012] Wende, C. (2012). *Language Family Engineering with Features and Role-Based Composition.* PhD thesis, Faculty of Computer Science, Technical University of Dresden, Germany.

[Wende et al., 2011] Wende, C., Aßmann, U., Zivkovic, S., and Kühn, H. (2011). Feature-based customisation of tool environments for model-driven software development. In de Almeida, E. S., Kishi, T., Schwanninger, C., John, I., and Schmid, K., editors, *Proceedings of Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011*, pages 45–54. IEEE.

[Wende et al., 2009] Wende, C., Bartho, Andreas Ebert, J., Jekjantuk, N., Gerd, G., Lemke, J., Miska, K., Rahmani, T., Sabina, P., Schwarz, H., Walter, T., Zhao, Y., and Zivkovic, S. (2009). D2.5 - Ontology Services for Model-Driven Software Development. MOST Project Deliverable - http://most-project.eu/documents.php.

[Wende et al., 2010] Wende, C., Thieme, N., and Zschaler, S. (2010). A Role-Based Approach Towards Modular Language Engineering. In van den Brand, M., Gaševic, D., and Gray, J., editors, *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 254–273. Springer Berlin / Heidelberg.

[Wikipedia, 2015] Wikipedia (2015). House of Cards. http://en.wikipedia.org/wiki/House_of_cards.

[Wikipedia, 2016] Wikipedia (2016). Digital Revolution. https://en.wikipedia.org/wiki/Digital_Revolution.

[Winskel, 1993] Winskel, G. (1993). *The Formal Semantics of Programming Languages: An Introduction.* The MIT Press.

[Wirth, 1978] Wirth, N. (1978). *Algorithms + Data Structures = Programs*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[Wittenburg et al., 1991] Wittenburg, K., Weitzman, L., and Talley, J. (1991). Unification-based Grammars and Tabular Parsing for Graphical Languages. *Journal of Visual Languages & Computing*, 2(4):347–370.

[Xu et al., 2010] Xu, T., Ma, W., Liu, L., and Karagiannis, D. (2010). Hybrid Modeling: Synthesizing Strategic Model and Business Processes in Active i*. In *14th IEEE international EDOC conference, Vitoria, Brazil*. IEEE.

[Zachman, 1987] Zachman, J. A. (1987). A Framework for Information Systems Architecture. *IBM Syst. J.*, 26:276–292.

[Zito and Dingel, 2006] Zito, A. and Dingel, J. (2006). Modeling UML2 package merge with Alloy. In *First Alloy Workshop*.

[Zivkovic, 2006] Zivkovic, S. (2006). *Integration Rules for Metamodel Integration in Business Modelling (In German)*. Master thesis, Faculty of Computer Science, University of Vienna, Austria.

[Živković and Karagiannis, 2015] Živković, S. and Karagiannis, D. (2015). Towards Metamodelling-in-the-Large: Interface-based Composition for Modular Metamodel Development. In *Enterprise, Business-Process and Information Systems Modeling*, pages 413–428. Springer.

[Živković and Karagiannis, 2016] Živković, S. and Karagiannis, D. (2016). Mixins and Extenders for Modular Metamodel Customisation. In *Accepted for International Enterprise Information Systems Conference, ICEIS 2016*.

[Zivkovic et al., 2007] Zivkovic, S., Kühn, H., and Karagiannis, D. (2007). Facilitate Modelling using Method Integration: An Approach using Mappings and Integration Rules. In Österle, H., Schelp, J., and Winter, R., editors, *European Conference on Information Systems, ECIS2007*. University of St. Gallen, Switzerland. http://is2.lse.ac.uk/asp/aspecis/20070196.pdf.

[Zivkovic et al., 2009a] Zivkovic, S., Kühn, H., and Murzek, M. (2009a). An Architecture of Ontology-aware Metamodelling Platforms for Advanced Enterprise Repositories. In *Proceedings of the 1st International Workshop on Advanced Enterprise Repositories (AER 2009), Colocated with 11th International Conference on Enterprise Information Systems (ICEIS 2009) Milano, Italy, May 6th*, pages 95–104.

[Zivkovic et al., 2011] Zivkovic, S., Miksa, K., and Kühn, H. (2011). A Modelling Method for Consistent Physical Devices Management: An ADOxx Case Study. In Salinesi, C. and Pastor, O., editors, *Advanced Information Systems Engineering Workshops - CAiSE 2011 International Workshops, London, UK, June 20-24, 2011. Proceedings*, volume 83 of *Lecture Notes in Business Information Processing*, pages 104–118. Springer.

[Zivkovic et al., 2015] Zivkovic, S., Miksa, K., and Kühn, H. (2015). On Developing Hybrid Modeling Methods Using Metamodeling Platforms: A Case of Physical Devices DSML Based on ADOxx. *International Journal of Information System Modeling and Design (IJISMD)*, 6(1):47–66.

[Živković et al., 2008] Živković, S., Murzek, M., and Kühn, H. (2008). Bringing Ontology Awareness into Model Driven Engineering Platforms. In Parreiras, F. S., Pan, J., Aßman, U., and Henriksson, J., editors, *Proceedings of the 1st International Workshop on Transforming and Weaving Ontologies in Model Driven Engineering (TWOMDE 2008), Co-located with MODELS, Tolouse, France, September 28, 2008*, CEUR Workshop Proceedings, pages 47–54. CEUR-WS.org.

[Zivkovic et al., 2009b] Zivkovic, S., Wende, C., Bartho, A., and Gregorcic, B. (2009b). D2.3 - Initial Prototype of Ontology-driven Software Process Guidance System. MOST Project Deliverable - http://most-project.eu/documents.php.

[Zivkovic et al., 2013] Zivkovic, S., Wende, C., Thomas, E., Parreiras, F., Walter, T., Miksa, K., Kühn, H., Schwarz, H., and Pan, J. (2013). A Platform for ODSD: The MOST Workbench. In Pan, J. Z., Staab, S., Aßmann, U., Ebert, J., and Zhao, Y., editors, *Ontology-Driven Software Development*, pages 275–292. Springer Berlin Heidelberg.

List of Figures

List of Tables

Appendix A

Biography

Srđan Živković is senior software architect at BOC Information Technologies Consulting GmbH. Srđan has many years of experience in developing ADOxx metamodelling platform and ADOxx-based modelling products such as ADONIS and ADOit. His work in software industry is complemented by the research in the areas of metamodelling, model-based approaches and tools, in which he also regularely publishes scientific articles. From 2008 to 2011, he participated in the FP7 EU research project MOST as a researcher and software architect. In addition, he was leading an industrial research project in semantic guidance for modelling language engineering co-sponsored by the City of Vienna (ZIT) and BOC. For several years, he's also been a regular adjunct lecturer in Business Informatics at the Faculty of Computer Science, University of Vienna.

Prior to his PhD studies at the Faculty of Computer Science, University of Vienna, Austria, Srđan Živković received a Master degree (Mag.) in Business Informatics (Wirtschaftsinformatik) from the University of Vienna, Austria in 2006 and a Bachelor degree (Dipl. Ing) in Information Systems from the University of Belgrade, Serbia in 2003.