Type-Safe and Conformance-Preserving Composition and Typing of Metamodels with Co-evolution of Models

Henning Berg

Thesis submitted for partial fulfillment of
the requirements for the degree of Philosophiae Doctor

Faculty of Mathematics and Natural Sciences
University of Oslo

June 2017

Abstract

Model-driven engineering is an approach that has increased in popularity during the course of the last couple of decades. In model-driven engineering models are first-class artefacts, which provides the software engineers with a powerful abstraction for handling complexity and ensuring separation of concerns. This allows engineering software that is easier to understand and reason about than traditional approaches to software engineering, which in turn lowers the costs of development and maintenance.

A key to the success of model-driven engineering is model management operations, e.g. model composition and model migration. However, current approaches for metamodel composition are not type-safe, or address how existing models are impacted when metamodels are composed. Ensuring type-safety and correctness of metamodel composition approaches, with following model migration and model composition, is of great importance.

This thesis describes 1) how the structure and operational semantics of metamodels can be composed and adapted type-safely while ensuring the validity of existing models, 2) how the operational semantics of metamodels can be integrated practically non-intrusively and 3) how metamodels can be typed to support variance and reuse.

Three main results are presented. The first result is a collection of language constructs that realise a type-safe template-based mechanism for composition and adaptation of metamodels, including a framework that migrate and/or compose existing models. The result also includes a framework for formal analysis of these operations. The second result is two mechanisms for integrating the operational semantics of metamodels, as defined by class operations, practically non-intrusively. The third result is a theory for realising metamodel types by means of class nesting, and how this allows substitutability and polymorphism for metamodels as a whole.

Acknowledgements

First and foremost, I would like to thank my main supervisor Birger Møller-Pedersen for his continuous support throughout my PhD study, for sharing his knowledge and for giving feedback on my work. I would also like to thank my supervisor Stein Krogdahl for always taking time for my questions and giving me feedback.

I am thankful to Ingrid C. Yu who learned me how to use formal methods and for being a co-author of one of the papers. Her knowledge in formal reasoning helped strengthen and solidify the work.

I would like to thank Eyvind W. Axelsen for helping me understand package templates to the degree necessary for elaborating on this mechanism for metamodels. I truly appreciate all the discussions with Gunnar R. Bergersen, Weiqing Zhang, Bjarne Holen and Kjetil Kjernsmo, and how they have always been helpful and supportive. I am also grateful for the discussions with Fritz Albregtsen.

A thanks goes to Øystein Haugen, Franck Fleurey and Arne J. Berre for evaluating and commenting on my work, and to the anonymous reviewers for giving me valuable feedback and suggestions for improvements.

Finally, I send my thanks to my colleagues in the Programming and Software Engineering group and the Precise Modelling and Analysis group for all the great moments and interesting talks we had.

List of Included Papers

1. Henning Berg, Birger Møller-Pedersen and Stein Krogdahl
   Advancing Generic Metamodels
   In proceedings of the co-located workshops on SPLASH '11:
   Domain-Specific Modeling (DSM'11), pages 19-24. ACM 2011.

2. Henning Berg and Birger Møller-Pedersen
   Type-Safe Symmetric Composition of Metamodels using Templates
   In proceedings of the 7th International Workshop on System Analysis and Modeling (SAM 2012), System Analysis and Modeling: Theory and Practice, Lecture Notes in Computer Science (LNCS), volume 7744, pages 160-178. Springer 2013.

3. Henning Berg and Birger Møller-Pedersen
   Specialisation of Metamodels using Metamodel Types
   In proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2014), Model-Driven Engineering and Software Development, Communications in Computer and Information Science (CCIS), volume 506, pages 84-99. Springer 2015.

4. Ingrid Chieh Yu and Henning Berg
   A Framework for Metamodel Composition and Adaptation with Conformance-Preserving Model Migration
   In proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2015), Model-Driven Engineering and Software Development, Communications in Computer and Information Science (CCIS), volume 580, pages 133-154. Springer 2015.

5. Henning Berg and Birger Møller-Pedersen
   Metamodel and Model Composition by Integration of Operational Semantics
   In proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2015), Model-Driven Engineering and Software Development, Communications in Computer and Information Science (CCIS), volume 580, pages 172-189. Springer 2015.

6. Henning Berg
   Migration of Models using Derived Transformations from Metamodel Adaptation and Composition Directives
   Unpublished. 2017.

List of Related Papers

1. Henning Berg, Birger Møller-Pedersen and Stein Krogdahl
   Application of Advanced Programming Concepts in Metamodelling
   In proceedings of Norsk Informatikkonferanse (The Norwegian Informatics Conference) (NIK 2011), pages 207-218. Tapir 2011.

2. Henning Berg
   Service-Oriented Integration of Metamodels' Behavioural Semantics
   In proceedings of the 7th International Conference on Software Paradigm Trends (ICSOFT 2012), Software and Data Technologies, Communication in Computer and Information Science (CCIS), volume 411, pages 155-170. Springer 2013.

List of Research Reports

1. Henning Berg and Ingrid Chieh Yu
   Generic Metamodel Refactoring with Automatic Detection of Applicability and Co-evolution of Artefacts
   Research Report 461, Department of Informatics, University of Oslo, Norway, 2017. ISBN 978-82-7368-426-4

2. Henning Berg
   Integration of Operational Language Semantics using Exported Namespaces
   Research Report 462, Department of Informatics, University of Oslo, Norway, 2017. ISBN 978-82-7368-427-1

Contents

Part I

Overview

Chapter 1

Introduction

Modelling is an essential activity in natural sciences. It allows representing a part of the world in terms of abstractions, which are easier to understand and reason about. This includes the ability to perform simulations of processes. In computer science, it may be argued that modelling is the very foundation that makes computing possible. In general terms, computer science is the discipline of creating models using computers. In many cases the models may also be executed for calculating or deriving new knowledge. Models in computer science are in principle mathematical models. They are expressed using a language containing modelling concepts often referred to as language constructs. A model or program comprises instantiations of the language constructs which are combined according to a set of well-formedness rules. The well-formedness rules are a formalisation of the language's structure and can be expressed in several ways, e.g. by using a grammar. A language's constructs reflect one particular problem domain, or domain under study.

A popular representation of a language's well-formedness rules is a metamodel. A metamodel is a precise formalisation of a set of valid models. And as the name suggests, it is a model of models. A metamodel defines the language of all the models possible to synthesise by using the metamodel. The most common way of defining a metamodel is in terms of a class model. The class model is either specified using a graphical or textual notation, where the former is the most common. A class represents a language construct, whereas the relationships between the classes define how the instances of the constructs may be combined to form valid models. Metamodelling is the activity of defining metamodels. It is a key activity in model-driven engineering [8] and domain-specific modelling [80].

The popularity of metamodelling has steadily increased since the invention and standardisation of the *Unified Modeling Language (UML)*[1] in 1997, even though the concept was discussed as early as in 1991 [81]. In particular, class diagrams or class models have been found to support metamodelling well. Hence, metamodelling is a rather young discipline and is a current research field. In most situations, a metamodel is considered as an alternative to a grammar, i.e. it defines the ab-

[1]http://www.uml.org

stract syntax of its models. However, various interpretations of what constitutes a metamodel exist. For instance, constructs for specifying static semantics, e.g. in the form of *Object Constraint Language (OCL)*[2] rules, may be considered being part of the metamodel. The *Kermeta* community[3] additionally considers operational semantics specified within class operations to be an inherent component of metamodels.

Software evolution is a driving force for change in computer science [82]. It reflects the need for updating software as new or altered requirements emerge or technological innovations present new opportunities. Additionally, performing general improvements and addressing errors in software drive software evolution. The term *software* covers all artefacts related to software development and use. This includes both models/programs and languages. Evolution of models[4] is something most users of software is familiar with. New versions of applications and operating systems appear on a regular basis. Languages need to evolve in a similar manner, though not as frequently. Specifically, the metamodel defining a language evolves in a process which is typically referred to as *metamodel adaptation*. Metamodel adaptation means to change (including extending and reducing) a metamodel for a specific purpose, e.g. to address changed or new requirements. The problem of evolving metamodels is considerably more intricate than evolving models. The reason is that there are bindings between a metamodel and other software artefacts. Consequently, changing a metamodel impacts all the artefacts that are defined relatively to the metamodel. This includes models, transformations, editors, model validators, analysis tools, etc. These artefacts, including the metamodel itself, constitute what is known as the *metamodelling ecosystem* [3]. The metamodel is the core artefact in any ecosystem, and changes applied to this need to be propagated to the related artefacts.

There are several model management operations/activities in model-driven engineering that can be applied to address evolutionary pressure. The three most common are model transformation, model composition and model migration. In principle, all operations can be explained and reified with model transformations. However, it is common to differentiate the operations for clarity and focus. The work of this thesis focuses on model composition (structure and operational semantics) and model migration. In addition, we investigate how metamodels may be typed.

Model composition is the operation of combining two or more models into a resulting model. The term covers combination of metamodels since a metamodel is a model. However, the activity of combining metamodels is typically more challenging than combining models because of the bindings between the metamodel and the other artefacts in the ecosystem, and therefore, involves additional sub activities

[2]http://www.omg.org/spec/OCL

[3]http://www.kermeta.org

[4]We use the word *model* to refer to both non-executable models and executable models/programs.

(e.g. model migration). To be precise, we use the nuance *metamodel composition* when referring to a combination of metamodels. Metamodel composition is applied to gain a holistic view of a system, e.g. for the purpose of verifying consistency or generating source code. Metamodel composition also addresses evolution by allowing to increase (and alter) the expressiveness of a metamodel. At a high level, metamodel composition is achieved by composing a metamodel with another metamodel or metamodel fragment resulting in a composed metamodel.

Metamodel adaptation is an activity that may occur in context with metamodel composition. It may be applied before metamodel composition to assist in the composition process or after to rectify undesirable effects resulting from the composition, or for the purpose of aligning the metamodel with the problem domain.

Model migration is the activity of co-adapting models as their metamodel is adapted due to evolutionary pressure. While model migration may be performed manually there are clear incentives for automating this process, which has resulted in an increased focus on mechanisms for model migration. The purpose of model migration is to change the models in such a way that they are compatible with the adapted metamodel. This is achieved by generating co-adapted models whose objects are all valid instances of classes in the adapted metamodel.

Model-driven engineering (MDE) envisions models as first-class entities. The relations in the metamodelling ecosystem reflect this vision. Specifically, artefacts relate to models at the abstraction level of models, and not in terms of their constituent objects. For instance, a transformation accepts models of a particular type; the same applies to model editors. Model typing is a research direction that aims at finding solutions for typing models as holistic entities to better support the model-centric view of MDE. Model typing allows to precisely specifying the input parameter(s) of an artefact in terms of a model type. The model type specifies properties that a model must have for it to be read and processed correctly by an artefact. This potentially means that models of different metamodels may be processed by the same artefact as long as they comply to the model type specified by the artefact. A derivative of model typing is metamodel typing. A metamodel type specifies properties that may be shared by several metamodels.

Meta(model) composition, model migration and model typing are ongoing research directions and there are still open questions that need to be addressed. Our overall goal with this thesis is to improve model management operations and model typing in MDE with particular focus on (1) supporting type-safe metamodel composition and adaptation including type-safe composition of operational semantics, (2) addressing model migration directly as metamodels are adapted and composed, (3) allowing metamodels to be typed using well-known object-oriented mechanisms, and (4) creating mechanisms for metamodel composition and model migration that are easier to understand and use than the current available mechanisms. The research theme underlying all contributions is composition, adaptation and reuse of metamodels and models.

1.1 Overview of the Mechanisms and Artefacts

The thesis addresses three main problems areas: metamodel composition (including composition of operational semantics), model migration and typing of metamodels. We have developed several artefacts during the work of this thesis. Each artefact contributes to one or more of the goals identified. The problem area(s) addressed for each artefact is mentioned.

1.1.1 Artefact A: Constructs for Type-Safe Metamodel Composition and Adaptation

We have defined an extension to the Kermeta metamodelling language with new constructs for composing metamodels' abstract syntax and operational semantics. The language extension also contains constructs for adapting metamodels to support a more precise or extended problem domain. The new constructs comprise a mechanism for type-safe metamodel composition with adaptation, which is based on defining metamodels as reusable templates, referred to as *metamodel templates*. Instantiating the new language constructs yields composition and adaptation directives which are applied on the metamodels given as input. The mechanism supports building hierarchies of metamodel templates. *Problem area: Metamodel composition.*

1.1.2 Artefact B: Framework for Derived Migration of Models

As a continuation of the metamodel template mechanism, we have defined a framework that utilises the user-specified composition and adaptation directives for migrating models. The directives are used to calculate a set of transformations that semi-automatically update existing models. The transformations can be generated instantly as metamodels are composed and/or adapted, or later when they are required. This includes the ability to selectively generate transformations between any two templates/metamodels in a template hierarchy that represent a valid migration route.

Problem area: Model migration.

1.1.3 Artefact C: Framework for Metamodel Composition and Adaptation with Model Migration

We have formalised a framework that utilises analysis for supporting migration of models as metamodels are composed and adapted. The framework takes one or more metamodels and an adaptation strategy as input. The adaptation strategy is analysed to verify that it can be applied successfully on the metamodels. The analysis calculates a set of effects which are used to alter the metamodels and transform existing models to re-establish model conformance. *Problem area: Metamodel composition and model migration.*

1.1.4 Artefact D: Framework for Non-Intrusive Integration of Operational Semantics

We have defined a framework that allows composing metamodels' operational semantics by defining mappings between metamodel constructs. The framework allows models of different metamodels to exchange information (at runtime) in a practically non-intrusive manner, thereby minimising the impacts on other artefacts in the metamodelling ecosystem. *Problem area: Metamodel composition.*

1.1.5 Artefact E: A Theory for Realising Metamodel Types

Metamodels may be typed by utilising class nesting. We have developed a theory that elaborates how this may be realised. The essence of the theory is how an enclosing class acts as a type for an enclosed metamodel. A metamodel may be adapted by utilising subtyping. Attributes and operation parameters may be typed with the enclosing class which consequently gives access to the metamodel structures. We refer to the enclosing class as a metamodel type. We have also studied how genericity can be achieved by supporting generic type parameters for the enclosing class. *Problem area: Metamodel typing.*

1.1.6 Additional Artefacts

We have developed two additional artefacts which we will only discuss briefly. These are a metamodelling language for integration of operational semantics and a theory for using the framework for metamodel composition and migration for refactoring of metamodels. The artefacts are elaborated in research reports and referred to as Additional Artefact I and Additional Artefact II.

1.1.7 Terminologies and Differences

There are some minor differences in terminology used in the papers and the thesis. We list them here to avoid confusion. First, the terms *adaptation* and *customisation* have been used to describe how metamodels are changed to address aspects of evolution. In this thesis, we use the term *adaptation* exclusively for this activity. Second, in some of the papers the term *dynamic semantics* is used. This term refers to what we in this thesis name *operational semantics*, i.e. executable semantics that is defined in/by class operations. Third, in the papers we use the phrase *integration of operational semantics* which refers to how the operational semantics of different metamodels are composed, e.g. by overriding operations. In this thesis, we will also use the phrase *composition of operational semantics*.

1.2 Structure of the Thesis

The thesis is divided in two parts. Part I introduces the work and puts it in context with established computing science disciplines and state of the art. It also presents the research method we have used and discusses the problems we have studied in terms of a problem statement and success criteria. We give a brief overview of each artefact and discuss and evaluate the work. This includes positioning the artefacts with respect to related work. Throughout the thesis we organise the discussion according to three overall areas of study: model and metamodel composition, model migration and model typing. Part II includes the research papers which document our work in greater details. The papers may be referred for additional information where necessary.

Chapter 2

Background

This chapter discusses the background for our work and presents important principles and practices on which we base our artefacts. We also discuss premises on which our problem statement is based.

2.1 Model-Driven Engineering

In essence, computing is the act of representing a real-life problem in terms of computer abstractions that represent a potentially executable model of the problem. The model may represent an intentional simplification of the problem. In the early days of computing, these models or programs were constructed in terms of abstractions of the solution space; the models included concepts related closely to the underlying computer hardware or operating system. In the 1960s and 1970s several programming languages were invented, in particular general-purpose object-oriented languages like Simula and Smalltalk. These languages provided language constructs on a higher abstraction level which meant that the models could now more closely reflect concepts of the problem space. Complexity became easier to handle and it was simpler to reason about the models and the problems being modelled.

The need for better abstractions, improved ways of handling the increasing complexity in software development and an increased demand for productivity are primary driving forces behind advancements in computer science. The languages of the 1960s and 1970s made programming simpler. However, manually writing code both requires a lot of resources and is error prone. An attempt to improve software engineering emerged in the 1980s in the form of *Computer-Aided Software Engineering (CASE)* [11]. CASE focused on using general-purpose graphical representations, e.g. state machines and structure diagrams, to model software solutions [9]. The representations could be used for generating implementation artefacts. CASE had some drawbacks hindering its wide adoption in industry, e.g. the general representations were too generic and could not be customised for specific application domains. Also, they did not map well to the underlying platforms.

Today, development using frameworks and platforms are prominent in industry. The frameworks and platforms provide common services for constructing applications in various application domains. However, complexity has risen to an almost unmanageable level which requires a lot of time being spent on implementation details and activities related to integration and deployment of software solutions [9]. The focus on low-level details also results in difficulties in gaining an overall understanding of a software system which reduces software quality and makes system evolution complicated.

Model-Driven Engineering (MDE) [8] is an approach that has increased in popularity during the last couple of decades. The principles of MDE can be traced back to the CASE effort, and it may be argued that CASE was indeed one of the first attempts of MDE. MDE alleviates complexity by allowing modellers to construct software at a higher abstraction level than what is typically possible using a traditional software development approach. This means that focus can be on the design and overall properties of software rather than on its implementation details. This in turn promises increased productivity and reduced costs.

Models are the primary artefacts in MDE and they are used in all engineering phases. Importantly, models are intended as machine-readable entities from which other artefacts may be synthesised using transformations or code generators. The models are typically domain-specific, meaning that problems are expressed using constructs that closely reflect concepts in the problem space. Low-level details are abstracted away from the models and instead incorporated in transformations or code generators that translate the models to executable programs, which may utilise pre-defined frameworks and platforms. A software system can be described using several models where each model represents a specific view(point)/concern or aspect of the system. This increases separation of concerns and allows different stakeholders to focus on their area of expertise.

MDE embraces models as the key artefacts. Appropriate tools for creating and manipulating models are essential in order to realise the MDE vision. Transformations and code generators are examples of such tools. Other types of tools includes model editors (concrete syntax), model checkers, tools for model differencing and versioning, and tools for composing and migrating models. These tools may also be referred to as modelling artefacts which realise model management operations.

There are two prominent approaches in MDE [27]. The first is centred around using UML with profiling to support modelling of domain-specific concepts (e.g. with the help of UML CASE tools). The second approach focuses on using dedicated *Domain-Specific Languages (DSLs)* which provide modelling abstractions that more closely reflect concepts of a particular problem domain. The software engineering practices differ depending on what approach that is chosen, and mechanisms for bridging profiled UML models and DSLs using model transformations have been devised, e.g. [27][28]. In the industry, it can be witnessed a movement from using general purpose languages (GPLs) to DSLs [18] which underlines the importance of

MDE. In this thesis we follow the second approach to MDE.

Model-Driven Development (MDD)[1] [14][10] is an approach that is often used synonymously with MDE. There is no accepted definition of how MDE and MDD relate. However, a common understanding is that MDD is more concrete than MDE, i.e. that MDE encompasses MDD[2]. Another name that often appears in association with MDE and MDD is *Model Driven Architecture (MDA)*[3]. MDA was standardised by the *Object Management Group (OMG)*[4] in 2001 and can be seen as one particular MDD approach. MDA specifies how software systems can be divided between high-level business models and low-level implementation models and thereby separating design from software architecture. One of the key operations motivating the usefulness of the separation in abstraction levels is the ability to perform model-to-model transformations, e.g. a model at a higher abstraction level can be transformed to a model on a lower abstraction level. *Model-Based Engineering (MBE)* is yet another name that appears in conjunction with MDE and MDD. It may be viewed as an approach in which models are utilised in software development, but where the models are not key artefacts as is the case with MDE and MDD[5]. In this thesis we will not differentiate between MDE or MDD, since the differences are irrelevant with respect to the mechanisms the thesis discusses. We will use the terms interchangeably.

2.2 Metamodelling

In MDE, the models are usually made using DSLs. A common approach for defining DSLs is to use metamodelling which yields one or more metamodels. A metamodel is a model of models [4]. That is, a model of a language of models [13]. A metamodel contains constructs for creating models within a specific problem domain (problem space) also known as the *System Under Study (SUS)*. Hence, the metamodel reflects the concepts and structure of this problem domain. In computer science, the use of the term *metamodel* typically corresponds to a class model. The origin of the class model is UML which was standardised by OMG in 1997. A class model has a set of classes which are connected by relations. It formalises the concepts and structure/relationships of a selected part of a problem domain. The resemblance to a grammar is clear, as a metamodel defines the abstract syntax its models may have. Moreover, both static semantics, e.g. in the form of OCL rules and operational semantics in the form of class operations [12] may be considered as inherent components of a metamodel.

The *Meta Object Facility (MOF)*[6] standard by OMG is considered the industry

[1]This is the same as *Model-Driven Software Development (MDSD)*.
[2]http://modeling-languages.com/relationship-between-mdamdd-and-mde
[3]http://www.omg.org/mda
[4]http://www.omg.org
[5]http://modeling-languages.com/model-based-engineering-vs-model-driven-engineering-2
[6]http://www.omg.org/mof

standard for specifying metamodels. MOF was coincidently standardised with UML in 1997. UML and MOF are aligned (MOF is a subset of UML) and both standards share the same metamodel for their definition. MOF is a metamodelling architecture oriented around the key concepts *Classifier* and *Instance* (or *Class* and *Object*). The properties of an instance are described by its classifier. Using classification results in metalayers where each layer is an abstraction of the properties of the layer right below in a linear manner. The MOF standard itself supports an arbitrary number of layers. However, in most literature four layers are typically considered to comprise the MOF architecture: the M0 layer contains the runtime data or runtime objects, M1 comprises models or programs whereas M2 contains the metamodel(s) with formalised concepts and structure for describing the models. Finally, M3 contains the meta-metamodel providing concepts and structure definitions for constructing metamodels on M2. The constructs on the meta-metamodel layer are meta-circular, which means that they can be used to describe themselves. Typically, the meta-metamodel is MOF itself or a derivative like the *Ecore* metamodel of the *Eclipse Modeling Framework (EMF)*[7].

Metamodels are defined during an activity known as metamodelling [2]. Metamodelling primarily comprises problem domain analysis, conceptualisation and abstraction which serve in the construction of one or more metamodels. A metamodel reflects the concepts and structure of one particular problem domain. A metamodel's concepts are usually defined at an abstraction level which makes it straightforward to map the metamodel concepts to the concepts and terminology of the problem domain. This opens up for non-technical stakeholders to take part in the modelling process and improves communication between non-technical stakeholders and the developers. *Linguistic metamodelling* is the most common form of metamodelling. It is typically used as a tool for defining languages [10]. MOF and EMF/Ecore support linguistic metamodelling.

An important notion in metamodelling is *model conformance*. Model conformance is a property which states whether a model is well-formed and valid according to a specific metamodel, which is also referred to as a reference model [4][32]. That is, whether a given model is a member of the set of all models that are possible to synthesise by using a specific metamodel. A model conforms to a metamodel if every object in the model is a valid instance of a class (and only one) in the metamodel. This is also known as *strict metamodelling* [5]. The model conformance property is reified as a relation between a model and a metamodel. Similar relations exist between editors, tools, transformations and their metamodel. And as mentioned, altering a metamodel thus impacts most artefacts in the metamodelling ecosystem [3].

There are alternatives when it comes to specifying domain-specific models. One of the most common approaches is to tailor the classes of a UML class diagram

[7]http://eclipse.org/modeling/emf

using profiles[8]. One of the key advantages using profiles is the many development tools that are available for UML. Another variant of DSLs is *embedded* or *internal* DSLs. An embedded DSL is a language that is implemented using general purpose constructs of a GPL with the purpose of extending the GPL with domain-specific constructs [83]. A GPL program using the embedded constructs is by definition a program of the DSL. In this thesis we only consider languages and DSLs that are defined using metamodelling.

Other Metamodelling Schemes

This subsection gives a brief overview of other approaches/architectures for metamodelling.

Multilevel Metamodelling MOF-based architectures are organised in four metalayers. However, a metamodelling architecture may have an arbitrary number of layers. An example of such an approach is discussed in [5], i.e. a way of organising UML where instantiation of model elements follows the same scheme regardless of model layer. For instance, instantiating a metamodel element should be no different than instantiating a model element. The approach builds on using strict metamodelling and *clabjects*. A clabject on one layer is an instance of a metaclabject on the layer directly above (except for the top layer). A clabject is an object that has both a type facet and an instance facet. The instance facet contains the attribute values (slots) and method instances defined by the metaclabject from which the clabject is an instance. The type facet of a clabject specifies properties that can be given values or instantiated, i.e. attributes and methods.

Deep instantiation is an alternative instantiation relationship that builds on the principle of clabjects [6]. The essence of deep instantiation is how model elements have a *potency*. The potency is an integer that describes how many times an element can be instantiated, e.g. an element with potency 2 can be instantiated twice, while an element with potency 0 can not be instantiated (and is e.g. an object or a slot, etc.). Instantiating an element decrements its potency by one. The approach also uses the notions of *single* and *dual fields*. A single field, e.g. an attribute, does only have a value if its potency is 0 (representing a slot). A dual field may have a value for each instantiation of the field. Using potency allows carrying information across model layers. For instance, an attribute can be "transferred" across layers if the potency of the attribute is 2 or higher.

Powertypes, Prototypical Concept Pattern and Nested Metalayers Other approaches for metamodelling include using *powertypes*, *prototypical concept pattern* and *nested metalayers* [6].

[8]http://www.omg.org/uml

A powertype allows defining a concept as an instance of one class and as a subtype of another. This makes it possible to provide features for the concept through inheritance, whereas the concept/object is still an instance of another class. In other words, both a direct and an indirect relationship are obtained.

Another way of combining instantiation and inheritance is to define a class as a subtype of another class (that provides a set of features). Both the class and superclass reside on the same layer. An object (on a lower layer) of the class will thus acquire the features of both the class and the superclass through instantiation. This is known as the prototypical concept pattern.

By differentiating between context, it is possible to achieve a strict *instanceOf* relationship for both M0 and M1 objects that instantiate classes at M2 (first-class relationships). Typically, this would not be strict as M0 objects are supposed to be instances of M1 classes. This is, however, possible by treating M0 and M1 as the same layer (nesting of layers), as seen from M2. At the same time an object at M0 may be linked to an object at M1 (second-class relationship). Hence, M0 objects may have a dual classification.

Ontological Metamodelling Ontological metamodelling takes metamodelling a step further from what a purely linguistic metamodelling approach is capable of. In an ontological metamodelling architecture it is possible to accurately describe how concepts at a given metalayer relate, i.e. how an entity is an ontological instance of another entity [10]. In other words, ontological metamodelling allows defining domain metatypes or metaconcepts which support a more accurate definition of a domain's concepts and their properties. The concepts are organised in ontological layers reflecting the ontological *instanceOf* relation between them. A framework named *MetaDepth* supporting both multilevel linguistic and ontological metamodelling is discussed in [7].

2.2.1 Metamodelling Frameworks and Tools

EMF[9] is one of the most popular frameworks for defining metamodels with generated tool support. It supports defining metamodels as Ecore models, conforming to the Ecore meta-metamodel, and their behavioural semantics in terms of Java code [86]. EMF supports automatic generation of skeletal Java classes, an API reflecting a specific Ecore model and a model editor. Each class in an Ecore model has a corresponding Java class and interface. EMF also supports reflective programming which makes it possible to program functionality (e.g. tools) in a generic manner, and thereby support reading and processing of models conforming to different Ecore models. EMF uses *XML Metadata Interchange (XMI)*[10] for persistence of Ecore

[9]http://eclipse.org/modeling/emf
[10]http://www.omg.org/spec/XMI

models and (instance) models. The Ecore meta-metamodel closely resembles *Essential MOF (EMOF)*[11] (Ecore models can be XMI serialised as EMOF models and vice versa).

Kermeta[12] is a workbench/metalanguage for building languages/DSLs [12][56]. It allows defining languages in a modular manner where the different engineering concerns, i.e. abstract syntax, static semantics and behavioural semantics, are expressed using dedicated meta-DSLs. The models of the meta-DSLs (at M2) are later composed using an open-class composition semantics (static introduction). Abstract syntax is expressed using EMOF, static semantics using OCL and behavioural semantics with Kermeta (using an imperative object-oriented action language). The behavioural semantics takes the role as operational or translational semantics. Kermeta allows building languages incrementally and supports language variants, e.g. different behavioural semantics may exist for a given metamodel. Kermeta is compliant with the Ecore metamodel.

Other metamodelling environments include the *MetaEdit+ Workbench*[13], the *Generic Modeling Environment (GME)*[14] and *eXecutable Metamodelling Facility (XMF)* [2]. *MontiCore Language Workbench*[15] is a grammar-based workbench for defining textual DSLs.

2.3 Model and Metamodel Composition

In MDE, a software system is typically modelled using several models where each model describes properties related to a specific viewpoint [20] or view [29]. The set of models (each confoming to its own metamodel) comprises a multi-model describing the entire system [110]. The main incentive for taking a multi-model approach is separation of concerns which leads to software development and maintenance that are easier to understand as complexity increases. This includes the ability to reason about the concern or aspect described by each model in isolation and validating each model separately [26]. Hence, a multi-model approach allows different stakeholders to focus on their expertise domain without having to relate to concepts outside of the domain, and increases productivity by providing more powerful language abstractions [2]. *Aspect-Oriented Modelling (AOM)*, e.g. [21], is an approach that relies on using a multitude of models to describe various system aspects. *Language-Driven Development (LDD)* is another approach that advocates using several languages in concert for software engineering [2].

Using a multi-model software engineering approach improves how a system is modelled and designed. However, the models eventually have to be combined to

[11]http://www.omg.org/mof
[12]http://www.kermeta.org
[13]http://www.metacase.com/mwb
[14]http://www.isis.vanderbilt.edu/projects/gme
[15]http://www.monticore.de

obtain a composed/integrated model of the system [20]. As an example, a system may be designed in several class models (different concerns) that are composed before implementation [92]. The composed model is required in order to combine the concerns/viewpoints captured by each model, e.g. for generating executable code or for verifying consistency. Model composition is also an important means of addressing software evolution, which is required due to e.g. changes in requirements and new technology. Another application of model composition is for supporting variability management in product line engineering [23].

Models can either be composed manually or (semi-)automatically. Manual composition of large models is tedious and prone to errors [29]. Composition is an activity that is used repeatedly in MDE, and there are clear incentives for performing composition automatically to the extent possible.

Model composition (or *model merging*) is the model management operation (or process) of combining two or more *source* (or *input*) models into a *composed* (or *output/target/result/composite*) model. Model composition approaches have emerged from work on aspect-oriented modelling, database scheme integration and model transformations [32]. Model composition may be understood according to three dimensions: *syntactic*, *semantic* and *methodic* [24]. The syntactic dimension covers how the concepts and structure of source models are combined into a new model. Semantic composition focuses on the meaning of the composed model in terms of the languages and composition mechanism used. This also includes the mental conception of what the composition of a given set of models means. The methodic dimension deals with composition seen in the light of development processes and tools.

At a conceptual level, model composition can be understood as addressing three concerns [20]. First, what concepts to be composed need to be identified. Second, the location in the composed model where the composed concepts are going to be placed has to be selected. Third, it has to be decided how the integration of the concepts should be performed. At a syntactic level, composition can be seen as an iterative process where concepts are composed successively [20]. The order in which the concepts are composed matters because elements reference each other. The process may be further divided in four phases: *the initial phase*, *comparison phase*, *merge phase* and *post-composition phase* [19]. In the initial phase the models are grouped according to their types. The purpose of the comparison phase is to identify equivalence between elements in the source models. This is determined by writing match rules that are executed by a match operator. The match rules refer to the model elements' signature. A signature is defined in terms of a model element's structural properties, i.e. an attribute or association end (reference) [25]. The models are combined in the merge phase using a composition/merge mechanism/operator. The merged model elements represent integrated concept views [29]. Finally, the composed model may be verified against well-formedness rules in the post-composition phase. Alternatively, the work of [20] proposes a model composition process with four phases. The process does not identify an initial phase as in [19] and has a phase named *conformance checking.* Conformance checking is used to identify potential conflicts between matching elements that disallow merging. In general, conformance checking may be integrated as part of the matching phase.

Model composition can be seen as a special type of model-to-model transformation [19][31][32][33]. Hence, model composition can be realised by using transformations [22]. In [33], the authors discuss different approaches for realising model composition by means of transformations. The approaches are differentiated based on how generic the underlying transformations are; from the dedicated transformation to the (theoretically) completely generic one. A dedicated transformation incorporates knowledge about a specific pair of models (primary and aspect) and only supports composition of these models. A transformation may support composition of a broader range of models at the cost of a more complex transformation algorithm. The task of the transformation is to incorporate the information of an aspect model into a primary model, hence achieving model composition. The opposite has also been argued for, i.e. that a model transformation is a special case of model merging [30]. The argument says that a non-empty model M_A may be merged with an empty model M_B, which can be seen as a transformation of the model M_A to a model M_C conforming to the target metamodel (i.e. $M_A + M_B \equiv M_A \rightarrow M_C$).

One attempt of presenting canonical definitions of the term *model composition* and requirements for model composition frameworks are discussed in [32]. In [24], the authors discuss a theory about the semantics of model composition in the form of an algebra. The motivation for the two works includes providing the necessary foundation for comparing model composition solutions and building new ones, and for assessing properties of existing solutions. In [32], composition is defined as the operation that takes two input models and combines them to an output model as dictated by the information in a correspondence model. The correspondence model contains links that specify how elements in the two models relate. Semantics for three types of composition operators is discussed in [24], i.e. *property preserving*, *fully property preserving* and *consistency preserving* operators. In general, a composition operator is defined as a function that takes two models as input and produces a composed model as output. A general semantic composition operator is also defined, i.e. as a function that takes two sets of systems as input and gives one set of systems as output. In [34], the authors formalise what yields conflicts in a signature-based composition of class models.

Metamodel composition is a special kind of model composition, where the composition is on a higher metalayer (M2) than that of model composition (M1). Metamodel composition typically requires additional management operations to be applied to ensure consistency and integrity, also referred to as coupled evolution or co-evolution, in other artefacts in the metamodelling ecosystem. (In this thesis we have focused on co-evolution of models, and not other artefacts that are impacted when metamodels are composed and adapted.) We differentiate between two main usage scenarios of metamodel composition: 1) composition of metamodels in a multi-view modelling approach where each metamodel reflects a specific concern or aspect, and 2) composition of metamodels with the purpose of increasing the expressiveness of a metamodel (including variability modelling/product line engineering). Both scenarios affect the existing models (if available) of the metamodels. For 1) this typically requires selecting existing models that are composed, whereas 2) may be addressed by composing existing models, or alternatively, update models to reflect the additional well-formedness rules of the composed metamodel. The latter situation occurs when there is not available models of all the metamodels used in the composition, e.g. when a metamodel is extended with a limited number of classes (metamodel fragment) that do not represent a concern that is typically modelled on its own. We argue that metamodel composition can also, in some cases (e.g. with respect to the second listed case), be seen as an evolution/adaptation, as a metamodel composition can alternatively be described in terms of added classes and class properties.

We primarily consider four methods, i.e. low-level operations, that can be used for composing metamodels. These are *merging of classes*, *creating a reference between two classes*, *using subtyping* and *using interface class(es)*. Additional methods are described in [99], e.g. *class refinement* and *template instantiation*. Merging of classes is achieved by taking the union of the contents of two or more classes in order to create a composite class that replaces the source classes. Creating a reference between two classes (of different metamodels) means that the one class may reference (and contain) one or more objects of the other class. Subtyping can be used to compose metamodels in three ways: a class of one metamodel may be created as a subtype of a class of another metamodel, or alternatively, the class may be made a supertype. Two metamodels may also be composed by specifying a class in each metamodel as a subtype of a common supertype. Interfacing of metamodels is achieved by creating one or more new classes that collectively act as an interface between two or more metamodels, i.e. the metamodels relate to the class(es) of the interface, e.g. by creating new references.

Model merging is another term used in in the literature for combining models. Model merging is a special case of model composition [32]. Model merging implies that all information from the source models are preserved in the resulting model, and that information is not duplicated in the resulting model. These requirements do not apply to model composition mechanisms which may have varying semantics, i.e. information duplicates may be allowed in the resulting model. The term *merging* is also used in the literature to describe how two model elements are combined or unified into one single model element, e.g. two properties with the same signatures may be merged into one property by a merging mechanism/operator.

Model weaving is an approach that is related to model composition. It can also be used as a basis for realising model composition [22]. Model weaving is the process of establishing links between models. The links are expressed in a *weaving* *model* (*correspondence model*). Hence, the weaving model can be understood as model composition at a more abstract level. The notion of weaving is also used in techniques for aspect orientation, i.e. for asymmetric weaving of aspect models into a base model, e.g. [34][31]. In this case, weaving is a specific approach for realising model composition.

Metamodels are adapted to address changes in requirements, platforms or as a consequence of new technology. Adaptation is typically achieved by manually altering the metamodel [58]. Adaptation of metamodels may also be performed as part of a metamodel composition. There are two types of adaptations that are common. First, it may be required to adapt metamodels to facilitate composition. A typical situation is to rename model elements to force or disallow matching between two or more metamodels in signature-based composition [29]. This is achieved by using pre-merge directives. Second, post-merge directives may be used to align the resulting metamodel to its target problem domain. This also includes addressing concerns regarding achieving a well-formed metamodel after the composition [19]. It should be noted that metamodels may also be adapted prior to composition for the purpose of aligning the resulting metamodel to its domain. However, we regard such adaptations as the result of applying post-merge directives since the order for which these adaptations are performed is typically of lesser importance.

The authors of [44] present a formal framework for proving the correctness of composition operators. The approach allows describing (compatible) higher-level composition operators in terms of two primitive composition operators: *union* and *substitution*. Models are treated as graphs. The union operator creates a combined graph from a set of source graphs. The substitution operator allows giving typed objects (graph nodes) new names, e.g. it is a form of renaming operator. Model conformance is assessed (expected property) and proven for both the operators. Construction of proofs for higher-level operators can be built on the proofs for the primitive operators. Building proofs for higher-level operators is based on verification of pre- and post-conditions (property-specific contracts).

2.4 Model Migration

Models capture business value, e.g. in the form of software solutions, business processes/logic and intellectual property. There is a clear incentive for ensuring that models are co-adapted and reused as their formal definition changes and evolves. Otherwise the models become invalid. This activity or process is known as *model migration*, *co-adaptation* or *coupled evolution/co-evolution*. In MDE, the term *model migration* means that models are updated to conform to an adapted or changed metamodel by executing a *migration strategy* [75]. Conformity is essential to ensure the inherent correctness of the models and that they can be understood and processed correctly by artefacts. A migration strategy can be specified in different ways, e.g. by using a transformation language, a GPL like Java or by combining specifically designed abstractions. *Exogenous* model transformations are typically used, as opposed to *endogenous* transformations [75]. The former type of transformation is defined between different types of metamodels, whereas for the latter type of transformation the source and target metamodel are the same. In MDE, a metamodel is expected to evolve [73], which underlies the importance of having support for model migration.

Co-adaptation of models (or other artefacts) can be described as a three-step process [3]: *relations definitions*, *change impact detection* and *adaptation*. Relations definitions means to specify the relations between the metamodel and the depending artefacts, e.g. the conformance relation between models and their metamodel. Change impact detection involves assessing impacts that occur in the models when changes are applied to the metamodel. The impacts depend on the relation being considered, e.g. the conformance relation. Adaptation means to update the models to valid instances of the target metamodel.

Model migration can be performed manually or semi-automatically by applying migration mechanisms. Manual model migration is a tedious and difficult activity which may lead to inconsistencies between a model and its metamodel [70][58]. It also requires a substantial effort which makes cost-effective MDE difficult to achieve [71]. As a result of these challenges, computer-aided model migration has been an active research direction the last decade.

There are three main types of model migration approaches [75][76]: *manual specification*, *operator-based approaches* and *metamodel matching approaches*. Manual specification means that every co-adaptation step is specified using a (dedicated) transformation language or a GPL like Java. Operator-based approaches allow specifying both metamodel adaptations and model migration using a library of coupled operators, i.e. a coupled operator both adapts a metamodel and specifies how models are to be migrated to accommodate the metamodel changes. Metamodel matching approaches infer migration strategies by analysing an evolved metamodel and its metamodel history [75]. There are two subcategories of metamodel matching approaches: *differencing approaches* and *change recording approaches*. Differencing approaches, also known as *state-based* approaches [18], work by analysing two metamodel variants with the purpose of deriving a difference model or matching model. The difference model may be created automatically or by application of heuristics that help identifying equivalences and differences between two metamodels. Which approach to use depends on the complexity of the metamodel adaptations. The difference model is used for inferring the migration strategy. Change recording approaches collect information about primitive metamodel changes and use these changes for inferring the migration strategy. Change recording approaches support a higher granularity of migration steps than differencing approaches. This is because analysing two metamodel variants may not reveal all the atomic migration steps that were applied to produce the evolved metamodel from the original metamodel, i.e. information required to create a correct migration strategy may not be available by comparing two metamodel variants. In [64] it is stated that evolution can be specified manually, be recorded or detected automatically for both operator-based and difference-based approaches. Metamodel matching approaches can be characterised as *dependent* since the requirements for evolution of models are inferred from the requirements for evolution of metamodels [117]. Similarly, approaches for manual specification are *dependent*. Operator-based approaches are *interdependent*; evolution of both metamodels and models are specified at the same time.

Metamodels can be changed/adapted in several different ways. This includes adding and removing elements, generalisation and restriction of properties, and applying refactoring operations like moving a property or extracting a class [58]. The type of change determines the complexity of the model migration and whether model migration can be achieved at all. It also determines how automatable the migration process is. In [72], metamodel changes are classified in three categories: *Non-breaking changes*, *breaking and resolvable changes* and *breaking and unresolvable changes*. Non-breaking changes means changes that do not break the conformance between existing models and the new metamodel variant. Adding an optional attribute (i.e. an attribute with a zero lower bound multiplicity) in a class is an example of such a change. A breaking and resolvable change is an adaptation that can be resolved and propagated automatically to the models, e.g. renaming a metamodel element is such an adaptation. Breaking and unresolvable changes can not be resolved automatically. That is, they break the conformance relation between the models and the metamodel. An example of such a change is setting a regular reference as containment, with the result that two objects may contain the same object(s). Another example of a change that is considered breaking and unresolvable is adding a property that has a non-zero lower bound multiplicity (i.e. a mandatory property) [57]. However, this change can be accommodated by generating a default value(s)/object(s) for the property in the model and thereby re-establish model conformance (though the user may typically later set/provide a correct value(s)/object(s) for the property to get a sound and meaningful model. By definition, the latter example is not automatically unresolvable. In this thesis, we only consider changes that disallow automatic re-establishment of model conformance as breaking and unresolvable changes. Breaking and unresolvable changes can typically only be addressed by human intervention because additional information (not possible to derive) has to be specified. A metamodel change may not always be possible to distinguish in terms of low-level atomic steps [70]. In such cases the order of the changes may matter because of interdependencies. Therefore, metamodel changes are further classified either as *parallel dependent* or *parallel independent* [70]. For parallel dependent changes interdependencies have to be identified and isolated.

Another way of viewing a metamodel adaptation and corresponding model migration is in the form of a *coupled change* [71]. Coupled changes are classified according to whether automatic migration is supported (and whether model migration transformations can be reused). Three categories are identified in [71]: *model-specific coupled change*, *model-independent metamodel-specific coupled change* and *metamodel-independent coupled change.* A model-specific change requires using a model transformation that can only be used to migrate one specific model of a given metamodel, i.e. the transformation can not be used to migrate other models of the same metamodel. A model-independent metamodel specific coupled change indicates that it is possible to create a transformation that can be applied on all models of variants of the same metamodel (i.e. between metamodels of a specific domain). Finally, metamodel-independent coupled changes represent adaptation and migration scenarios that can be expressed in a generic manner independently of a specific domain/metamodel variants.

Metamodel adaptations can be categorised according to their preservation properties, i.e. different forms or levels of *semantics-* and *instance-preservation* [58]. The different levels of instance-preservation describe how instances of a metamodel are preserved or varied during migration. Instance-preservation is correlated with semantics-preservation. Based on the level of semantics-preservation, metamodel adaptations can be grouped in *refactoring*, *construction* and *destruction.* Alternative terms used to categorise adaptations are *additive* or *additions*, *subtractive* or *deletions*, and *updative* or *changes* [57][74].

2.5 Model Typing

A type can be understood as a collection of computational entities that share one or more properties [77]. Types allow naming and organising concepts and ensuring that data is interpreted and processed in a certain manner. If a model or program is type-safe it can be guaranteed that the data contained in the model or program is manipulated, e.g. by operations, within the boundaries defined by the types. By relating types, e.g. in the form of subtyping, it is possible to support substitutability, reuse, customisation and genericity. Typing is a well-known concept in programming and modelling languages, e.g. object-oriented languages use classes for explicitly defining the types of the model objects.

A model in MDE is typically represented as a collection of objects. Each object has a corresponding class in a MOF-compatible metamodel [53]. The classes of the metamodels are related using bi-directional references and subtyping. These relations are reflected in the models, e.g. a reference between two classes yields a link between two or more objects of the respective classes. In MDE, models are considered first-class artefacts. Other artefacts in the metamodelling ecosystem read and process the models, e.g. transformations and editors. From an architectural point of view it is unnatural to think about the type of a model in terms of the types of the objects it contains, as the other artefacts in the ecosystem intuitively accept models as input and not a collection of objects [53]. Instead, it should be possible to reason about the type of a model as seen as a holistic entity. There has not been much work on model types in MDE. The most prominent work is based on type matching using a structural conformance relation which indicates whether two models are conformant [53]. That is, a model type matches a reference model type if the model type have all the metaclasses of the reference model type (with identical names), and the classes of the model type contain the same number of properties and operations with identical signatures as the respective classes in the reference model type. This means that the model type must have at least the same structure as the reference model type.

A related topic to model typing is *megamodels* [78]. A megamodel is a model conforming to a metamodel with elements representing models, metamodels and other artefacts. This means that models and artefacts can be related at a higher abstraction level which gives an architectural view on the MDE ecosystem. Also, *family polymorphism* [79] is related to model types. Family polymorphism is a programming language mechanism that supports expressing and managing multi-object relations, i.e. class families. A family is characterised by a set of class relations. The actual classes are not known statically, i.e. it is a generalised kind of polymorphism.

Chapter 3

State of the Art

In this chapter we discuss the state of the art of (meta)model composition, model migration and model typing.

3.1 Model and Metamodel Composition

There exist many different approaches to model composition. In this section we go through the most important ones, and categorise them according to their operational scheme for easier reference. We use the six categories: *signature-based and rule-based composition*, *aspect-oriented model composition*, *other composition approaches*, *composition by establishing semantic links*, *integration utilising ontologies or semantic analysis* and *component-based integration.* For completeness, we include two UML-based mechanisms for composition and adaptation.

Package Merge and UML Profiles

One of the simplest approaches to model composition is known as *UML Package Merge*[1]. Package Merge is a name-based composition approach that works by merging identically named classes of different packages. Specifically, the mechanism is used to combine the different views of the UML metamodel. The mechanism can be used iteratively to construct larger packages from smaller ones.

Another mechanism provided by UML is *profiles.* Profiles support extending and restricting UML metaclasses with the purpose of better reflecting a problem domain or a platform. They are defined using stereotypes, tag definitions (a stereotype's properties) and constraints. Profiles allow the modeller to e.g. provide a syntax and notation for concepts, new types and semantics.

[1]http://www.omg.org/uml

3.1.1 Signature-Based and Rule-Based Composition

Model Composition using Kompose

A straightforward approach for signature-based model composition is supported by the *Kompose* tool [26], as implemented in Kermeta. Kompose allows decorating a language's metamodel with functionality for automatic matching and merging of elements in models conforming to the metamodel. The purpose of the decoration is to identify classes in the metamodel whose instances comprise mergeable elements, and to specify signatures for the model elements. The latter also includes defining what makes two signatures identical with respect to merging. A signature (or signature type) is a set of syntactic properties associated with a model element (type) [29]. The signature distinguishes one model element from the others, i.e. two elements with the same signatures can not coexist in a model. Merging of model elements is achieved by using a generic composition operator. The algorithm for the composition operator may be redefined e.g. to get a specific order for the composition of the model elements. Kompose includes a mechanism for conflict detection. It also allows the user to specify pre- and post-directives for handling conflicts, forcing or disallowing matching between elements, overriding default merge rules and adapting the composed models. Kompose also supports matching and merging of Ecore-based metamodels. We see Kompose as a signature-based composition approach. However, it can also be viewed as a simple aspect-oriented composition mechanism (that utilises a signature-based composition scheme).

Model Merging with EML

The *Epsilon Merging Language (EML)* [30] allows merging models using rules and strategies. EML is built on top of the *Epsilon Object Language (EOL)*[2]. Match rules are used to describe what gives a valid match and equality between classes of different metamodels. Merge rules specify what elements that can be merged and the elements that are produced in the target model resulting from the merging. Finally, transform rules describe how elements in the source models, which do not have a match, are transformed to elements in the target model. The specified rules are tested iteratively with respect to two source models. Strategies are algorithms that allow defining the matching, merging and transformation logic in a generic manner. Strategies are particularly useful in situations when models are instances of the same metamodel by reducing the number of rules that have to be specified explicitly. Strategies can be inferred automatically [32].

[2]http://www.eclipse.org/epsilon/doc/eol

3.1.2 Aspect-Oriented Model Composition

Model Weaving using XWeave

XWeave is a model weaver that allows asymmetric weaving of aspect models into a base model [31]. It is based on EMF and allows weaving both metamodels (Ecore models) and models (instances of Ecore models). Model elements intended to be woven together (pointcuts) are identified by matching names or by defining expressions in the *openArchitectureWare (oAW)* expression language[3] (a derivative of OCL). Both homogeneous and heterogeneous aspect models are supported, i.e. aspect models whose elements are added several places in the base model or only one place, respectively. The term *weaving* used by the authors reflects how aspects are woven with/into a base model in the same way as weaving is used in aspect-orientation techniques. This includes specifying pointcuts. However, the very mechanism of merging an aspect model with a base model corresponds to how the term *model composition* is used by (other) model composition approaches. XWeave only supports adding model elements to a base model (positive variability) and not removing elements (negative variability).

In [52], the authors discuss an approach for aspect-oriented and model-driven product line engineering. XWeave is here used to support positive variability. The authors also discuss a tool for negative variability, named *XVar* and a tool for defining relations between metamodels known as *XJoin*. XWeave, XVar and XJoin are tools in oAW.

Generic Weaving with GeKo

GeKo is an aspect-oriented tool for weaving advice models with a base model as dictated by a pointcut model [38]. The tool is built using Kermeta and thereby supports all types of models that are instances of EMOF metamodels. The approach works by defining mappings between the base, advice and pointcut models. The mappings are realised as links between the concrete syntax of model elements. A pointcut model is either a subset of the base model or defined in an abstract manner by giving metamodel elements roles that can be substituted by join points in a base model. The latter approach supports defining reusable aspects which may be utilised for different base models (e.g. security and authentication aspects). Matching of pointcuts with base models for the purpose of identifying join points is achieved using a Prolog-based matching engine. Composition of models works by *partitioning* the objects of the base and advice models in five sets and later *cleaning* the resulting model. The base model is partitioned in three sets: objects that will be kept as is, objects that will be removed from the model and objects that will be replaced by advice model objects. Similarly, the advice model is partitioned in two sets: objects that will replace base model objects and objects that will be

[3]http://www.eclipse.org/gmt/oaw

added to the base model. The partitioning is induced by the mappings (morphisms) between the base, aspect and pointcut models. Hence, composition is achieved by replacing base model objects with advice model objects. Cleaning consists of two steps. First, the properties in an advice model object, that replaces a base model object, are modified with information from the corresponding properties in the replaced base model object. In other words, primitive values and links are maintained by the replacing advice model object. Also, other base model objects (that will be kept) linking to replaced objects are updated to point to replacing objects in the advice model (as long as the respective reference is not composite). Second, links to removed objects in the base model are removed to get a consistent resulting model. This includes removing an object that links another object(s), corresponding to a property with a lower bound of 1, that has been removed during partitioning.

Reusable Aspect Models

In [46], the authors discuss an approach for aspect-oriented design of software systems in terms of aspect models. An aspect model describes both the structure and behaviour of a specific concern and comprises three UML diagrams: a class diagram, a state diagram and a sequence diagram. An aspect has an interface consisting of all the public operations in classes in the class diagram. The sequence diagram describes how objects collaborate at runtime, whereas the state diagram indicates what messages an object accepts in its various states. The term *aspect* is in aspect-oriented approaches typically used to describe a cross-cutting concern. However, in [46] an aspect is a reusable piece of functionality that addresses any kind of concern. Aspects are composed using two composition mechanisms. Class diagrams are merged using Kompose [26]. State and sequence diagrams are woven by detecting join points according to a pointcut model, which serve to identify where an advice model is applied/composed. Any model element can be used to define a pointcut and hence serve as a join point. An aspect may depend on other aspects, which gives a hierarchy of aspects, i.e. a high-level functionality aspect can be built from low-level functionality aspects. An important consideration is that aspects should be general and reusable for several applications. This implies that systems are decomposed into several low-level functionality aspects. The approach does not support automatic detection of conflicts between aspects. However, warn0ings about potential conflicts are issued to the modeller. Conflicts may be resolved by defining *conflict resolution aspect models*. These models specify a conflict criteria condition that states when a conflict occurs and the adaptations required to address the conflict. The adaptations are applied automatically when the conflict criteria condition is verified. An aspect is used in a target model by instantiating it. This involves mapping instantiation parameters (e.g. classes) of the aspect to elements in the target model.

Composing Multi-View Aspect Models

In [34], the authors discuss how multi-view aspect models are composed. The work builds on the *KerTheme* approach [87], which allows describing a concern in a model with two views. The first view is expressed using *Executable Class Diagrams (ECD)*. An ECD models the concepts and structure of its problem domain and the behaviour of the classes (objects) reflecting these concepts. The second view is modelled with sequence diagrams and models the global interactions and coordination between objects in models conforming to the ECD. This view is also referred to as a scenario. A model either represents a base or aspect concern, i.e. either a base or aspect model. Composition of two base models (both views) is achieved using symmetric merging, whereas composition of a base model with an aspect model (both views) is realised using asymmetric weaving. Merging combines two model elements that represent a different view on the same concept. The authors propose four composition operators that support the various composition scenarios.

Two base ECDs are merged using a signature-based approach. The composition process consists of finding matching model elements in two source models based on the model elements' identifiers (names). Matching model elements are then compared to detect conflicts, e.g. merging two classes containing equally named properties with different types yields a conflict. Conflict resolution is performed by manually providing composition specifications, in the form of transformations, that address how the conflicts should be resolved. Composition of base ECDs include combining classes, properties and operations, respectively. Merging of operations is achieved by renaming the matching operations (i.e. operations with the exact same signature) and defining a new operation that invokes these renamed operations in a specific order.

Composition of the sequence diagrams in two base models mainly works by manually identifying points at which the one diagram is merged with the other. Three merge operators are suggested by the authors: *amalgamated sum*, *sequential composition* and *inclusion*.

Composing the ECD of a base model with the ECD of an aspect model is achieved using aspect weaving. Aspect weaving has two phases. First, a part of the aspect model, known as the pointcut, is used to determine join points, i.e. places in the base model where the aspect model should we woven in. Second, another part of the aspect model, known as the advice, is composed with the identified join points. The join points, i.e. join point operations, are described using an expression language. The advice is a class that contains three crosscutting operations (inherited from a special purpose aspect class): _pre, _proceed and _post. The operations specify crosscutting behaviour. The advice/aspect class is renamed and merged with every base join point class that contains a join point operation (i.e. several copies of the advice class are used if needed). Conflicts occur between the operations in the join point class and aspect class by giving operations identical names. The conflicts are resolved in two phases by the merging operator which gives a correct order regarding how the operations should be invoked. Consequently, the crosscutting behaviour is implemented in the base model.

Asymmetric composition of sequence diagrams is achieved by specifying an aspect as two sequence diagrams; one sequence diagram comprises the pointcut whereas the other specifies the advice. The composition works by replacing the behaviour at the detected join point in the base model with the behaviour of the advice in the aspect model.

Aspect Weaving Utilising JPDDs

Aspect-orientation for MDD is taken a step further in [35]. The approach alleviates how complex pointcuts can be specified by allowing the designer to work at a higher abstraction level. In particular, the approach improves how to deal with situations of dynamic join point selection constraints, i.e. constraints that are based on runtime information and decide whether certain (advice) code should be executed. The approach works by weaving an aspect model into an application model (base model), according to pointcut descriptions using *Join Point Designation Diagrams (JPDDs)*. A JPDD is a UML profile which allows specifying queries, i.e. pointcuts are specified using a (profiled) sequence diagram. Contrary to other MDD aspect-oriented weaving approaches, the approach of [35] does not work by replacing a matched model segment with an advice. Instead, a set of automatically generated model transformations are responsible for finding potential join points. The transformations are also responsible for collecting runtime information required to evaluate the dynamic selection constraints and verifying that an advice is executed only if the dynamic selection constraints are satisfied. The transformations perform changes at several places in an application model. The weaver is based on a set of transformation templates which are instantiated with values specified in the JPDD given as input to the weaver. The templates represent patterns for monitoring the dynamic conditions: *control flow constraints*, *state-based constraints* or *data flow constraints*. Using a JPDD makes it easier to specify pointcuts whose dynamic join point selection constraints are based on information that is scattered in the base model. It also relieves the designer from manually having to verify the consistency between the model transformations as a result of evolutional changes, (i.e. due to dependencies between the transformations). The approach allows using a different notation for specifying pointcuts than the notation used for the base and aspect models. The result of the weaving is a UML model. The authors of [35] state that the benefits of their approach are at a minimum if the information required to evaluate the dynamic selection constraints of a pointcut is gathered at the point where the aspect is going to be applied (i.e. if the information is not scattered in the base model).

Aspect-Oriented Composition using Graph Transformations

An approach for aspect-oriented composition using graph transformations is discussed in [40]. Base models are described using UML. That is, class, sequence and state diagrams are supported. An aspect is described as an increment of a base model or another aspect; it describes both join points (in the form of pointcuts) and advices, i.e. how the base model should be modified at the join points. Any type of model element can be a join point and advices do not have restrictions, e.g. they are not limited to the standard *before*, *after* and *around* advice types. Pointcuts are described as a sequence of elements which allow identifying join points precisely (only available for state and sequence diagrams). Aspect rules are expressed in the concrete syntax of the UML diagrams (it is common to specify such using abstract syntax). Critical pair analysis (a way of detecting interactions between graph transformation rules) is used to detect conflicts and dependencies. A conflict indicates that an aspect prevents another aspect from being applied (which requires the base or aspect models to be modified), whereas a dependency indicates that one aspect has to be applied before another. Code generation from class and state diagrams is supported.

Contract-Based Composition

Obliviousness is a desirable property in aspect-oriented approaches, in which base models are created unawarely of aspect models. The authors of [45] motivate that obliviousness may break the interfaces of a base model and important assumptions made by its creator. This is because the application of aspects may alter the base model in ways not foreseen or desired by the creator of the base model. To address this concern, the authors present *contract-based composition* of model-based aspects. A *contract* acts as an interface of the base model dictating what elements that can be accessed and changed by an aspect. The contract can be divided in two parts: a *composition contract* and an *assumption contract.* The composition contract specifies what elements of the base model that can be accessed and modified when applying an aspect. The assumption contract is derived from an aspect and describes the intention of the aspect, i.e. how it will affect a base model. The two contract parts are compared to check the applicability of an aspect on a given base model. The composition contract is specified manually and constitutes an instance of a metamodel comprising the contract concepts. The constraints in the composition contract are specified using an extended version of OCL. It is also possible to specify invariants which are checked for the composite model. The approach is compatible with MOF-based languages. The purpose of using contracts is to prevent illegal modifications of the base model and reduce the chance of introducing semantic errors.

Theme/UML With MDA Support

In [50], the authors discuss an approach where MDA is realised on top of *Theme/UML*. Theme/UML is an aspect-oriented language created as an extension to UML. It provides constructs for modularisation and composition. The central unit of encapsulation is a *theme*. The theme construct is an extension of the UML package concept and is used to specify a base or aspect concern. A theme can include any type of UML diagram. An aspect theme utilises templates. The interaction of the templates with a base theme is described using sequence diagrams. A theme is a self-contained unit which does not refer elements outside of its definition, i.e. it is declaratively complete. Composition is performed by either *merging* elements, *overriding* an element with another or by *binding* an aspect theme to a base theme, i.e. for composition of a crosscutting concern with a base theme. Binding supports merging of both the syntax and behaviour of the aspect theme with the base theme. The authors in [50] use Theme/UML as basis for realising MDA. In their approach, the composition semantics of Theme/UML is defined by a marking profile. (Marking allows identifying elements for transformation non-intrusively.) Using this profile allows a designer to create a composition specification. A UML diagram and a file containing marked UML elements are mapped/transformed to a model which is an instance of a composition metamodel (specified using EMF). The composition model is transformed (i.e. composition of the themes/models is performed) to a platform-independent model and later to a platform-specific model (refinement). Finally, the platform-specific model is used for code generation (synthesis). The transformations are described using Java and *XPand*[4] (a template language).

3.1.3 Other Composition Approaches

Conceptual Domain Composition

An alternative take on composition of DSLs is discussed in [36]. The approach builds on what the authors refer to as *conceptual composition*, i.e. the domains of DSLs are combined by creating a composed domain. The composed domain is hence a derivative of the subdomains of the constituent DSLs. Composition is reified at three different places. First, a composed metamodel is defined by building on the metamodels of the DSLs and creating relations between their concepts (i.e. classes). The composed metamodel reflects a composed domain of the DSLs' subdomains. Two types of relations may be used: *associations* and *correspondences*. An association relates concepts that have different semantics. A correspondence relates concepts that represent a unified concept in the composed metamodel, i.e. the concepts represent different aspects of a unified concept and have overlapping semantics. Hence, the composed metamodel encompasses the metamodels from two or more DSLs with added relations between their concepts. The composed metamodel

[4]http://www.eclipse.org/gmt/oaw

may also be extended with new concepts and relations to reflect an extended domain. Second, to achieve integration of the DSLs behavioural semantics, a new composition interpreter has to be implemented. The composition interpreter's task is to synchronise the interpreters of the DSLs. Aspect-oriented programming is used to capture events in the DSLs' interpreters which results in invocations of methods in the composition interpreter. Third, a composed model is created for relating the existing models made using the DSLs. The composed model conforms to the composed metamodel. It is interpreted by the composition interpreter. The composed metamodel and composition interpreter give a new DSL which can be composed with other DSLs. An important design decision is that existing metamodels, models and interpreters (and other tools) do not need to be changed (inspiring the conceptual nature of the composition approach), i.e. as a consequence of evolution. The new artefacts act as a superstructure on top of the existing artefacts. The interpreters are implemented in Java. Each metamodel concept/class has a corresponding class in its associated interpreter. The concept's behaviour is defined as methods in the interpreter class. This allows treating the state of an interpreter (i.e. a set of Java objects) as the runtime state of the modelled system[5]. Mappings are defined between an interpreter and tools using aspect-oriented programming. The tools interface towards the "real" system being modelled and allows synchronising the state of the interpreter/model with the state of the system. The approach allows using features which are implemented similarly as mappings. A feature allows defining optional domain behaviour that can be selected for certain applications in the domain. That is, an application can be adapted to address additional or alternative requirements. This is possible without changing any of the existing artefacts.

Composition through Parameterisation

In [37], the authors discuss a composition framework for assembling DSLs from smaller building blocks referred to as *domain concepts*. A domain concept comprises a metamodel and a transformation. The transformation maps models of the domain concept metamodel to models of one or more target metamodels/languages whose semantics is well-defined. Hence, the semantics for a domain concept is defined in terms of a transformation to another language (or languages). Composition of DSLs is achieved by parameterising a domain concept with another domain concept, i.e. by substituting a domain concept with another domain concept, either partially or totally. Parameterisation is applied at both a syntactic and a semantic level. At the syntactic level, the parameterisation allows composing the metamodels of different domain concepts. This is achieved by specifying two types of parameters; a *formal* parameter is substituted by an *effective* parameter according to a (possibly empty) set of conditions. A parameter is a subset of a metamodel, i.e. a selected set of classes and relations. This means that only a few selected classes and relations from

[5]This resembles how EMF and Kermeta work.

a metamodel may be used as an effective parameter while the remaining classes and relations of the metamodel are discarded. Parameterisation at the semantic level concerns transformation composition. The approach supports both composition of DSLs and adaptation of DSLs.

3.1.4 Composition by Establishing Semantic Links

Model Weaving using the ATLAS Model Weaver

Model weaving is an approach for establishing relations between the elements of different models using *links* as expressed in a weaving model [22]. A link has a type that reflects the semantics of the relation it represents, e.g. *equivalence (merging)*, *replacement (overriding)*, *subtype (inheritance)*, etc. Model composition is one of the applications of model weaving. It is realised in two steps. First, elements of different models are related in a weaving model. Second, the weaving model is then used to generate model-to-model transformations that are executed to compose the models. Using weaving models allows defining model composition at a more abstract level by hiding away details on how the composition is realised. That is, the specific details on how the composition is performed are implemented in the transformation generator and the transformations. A weaving model can be created manually or semi-automatically by executing matching heuristics with manual refinement. As a comparison, executing match rules using EML gives a weaving model [32]. The weaving model is also known as a correspondence model.

The authors discuss an approach for semi-automatically generating weaving models and model transformations in [55]. Matching transformations are executed with the purpose of producing a weaving model. The weaving model is refined using matching heuristics (calculating similarity values between elements) and manual intervention. The best fitting links are selected using *filtering* whereas *rewriting* analyses the relationships between links to capture transformation patterns, e.g. nesting, inheritance and concatenation. The final version of the weaving model may then be used to generate model transformations.

Integrating Models Through Mega Operations

The authors of [41] discuss a related approach to model weaving where MOF-based metamodels and models are integrated. Two types of integration operations are motivated: *weaving* and *sewing*, referred to as *mega operations*[6]. Weaving addresses scenarios where different concern/aspect models are assembled to produce a model for a whole domain. Hence, weaving supports a tight integration of a set of models. The operation works on both the metamodel level and the model level. Specifically, a model conforming to one metamodel is woven with a model conforming to

[6]A megamodel is a model whose elements represent models.

another metamodel yielding a model conforming to the woven metamodel (resulting from weaving the two source metamodels). Weaving is further differentiated in four operations: *overrides*, *references*, *prune* and *rename*. The *overrides* operator allows replacing conceptually (semantically) overlapping model elements, *references* supports adding associations between classes, *prune* makes it possible to remove redundant model elements, whereas *rename* can be used to give model elements new names. Despite using the word *weaving* by the authors, this kind of operation is typically referred to as model composition in the literature.

Sewing takes another approach for integrating models. The sewing operation supports scenarios where different domain models need to be integrated while at the same time preserving their autonomy. In contrast to the weaving operation, sewing does not combine the metamodels and models intrusively. Instead, the operation utilises *mediators* which dictate the valid sewing possibilities at the model level. Sewing is divided into the operations *synchronizes* and *depends*. The *synchronizes* operation e.g. allows propagating values between class attributes in different models, whereas the *depends* operator makes it possible to specify structural dependencies, i.e. that a model element in one model depends on a model element in another model. The authors note that mediators on the model level, corresponding to the *synchronizes* and *depends* operations, may be realised using *Query, View and Transformation (QVT)*[7] transformations. The authors also discuss how alternative mediators representing associations and generalisations may be achieved using *Java Metadata Inferface (JMI)*[8] and EMF. That is, the mediators are established in terms of runtime Java objects. On the code level, the authors suggest that executable code generated from the (structural) models can accommodate the mediating behaviour (in the form of glue code) of the sewing operation either by manually customising the generated code or by using aspect-orientation, e.g. based on the *AspectJ*[9] weaver. Using web services to achieve the mediating behaviour between models is also suggested.

Both the weaving and sewing operations are elaborated with integration constraints which act as filters to further constrain what model elements that should be affected by the operations. As an example, an element may only be overridden by another element if both elements in question have the same value for an identifier attribute.

The authors discuss how the weaving and sewing operations can be realised as QVT transformations, though they do not state having verified their approach by the implementation of a prototype. An important motivation for their approach is reuse of integration knowledge by addressing integration at both the metamodel level and the model level, i.e. not only at the model level. The authors state that integration at the model level should be performed fully automatically by deriving (and execut-

[7]http://www.omg.org/spec/QVT
[8]http://jcp.org/aboutJava/communityprocess/final/jsr040/index.html
[9]http://eclipse.org/aspectj

ing) integration directives, as governed by the integration at the metamodel level. Another important motivation for the work is that specifying basic transformations manually do not scale up for complex models. Instead, integration can be handled by abstractions in the form of weaving and sewing operations.

Relating Views Using a Multiple-View Metamodel

An approach for variability management in product lines is described in [51]. The approach builds on unifying different phases in software development and views within these phases by expressing relations in a multiple-view metamodel. A phase is modelled as a composite class composed of different view classes. Each view class is again a composite class comprising classes representing concepts in the view. In other words, classes are nested in two levels. View consistency is enforced by checking rules defined relatively to the relationships between the classes (of different views) in the metamodel.

3.1.5 Integration Utilising Ontologies or Semantic Analysis

Integration of DSLs Using Ontologies

One way of integrating languages is by unifying these in terms of an ontology. The authors of [39] present one such approach where a reference ontology is used for defining semantics for metamodel constructs. Integrating several languages involves identifying metamodel constructs that overlap semantically. By mapping the relevant/identified classes and their properties of the metamodels to the concepts and relationships of a common ontology it is possible to express cross-language relationships and constraints. That is, semantic links are established between models of different languages. The approach allows representing model elements in terms of ontology concepts and property assertions (representations that can be used to derive semantic equalities between models) instead of merely describing the correspondences between elements in different models which is the case using a weaving model (e.g. equivalence, subtype, etc.). The authors present an upper ontology of general concepts for modelling languages relevant to system design and implementation. The authors state that more specific ontologies would also be required. The motivation for using an ontology to integrate languages is to achieve a decoupled integration of different models (and the models are kept consistent), the ease in which ontologies can be extended with application specific knowledge, and support for reasoning and inference. Also, using an ontology gives a holistic view of how modelling artefacts relate in an overall system description. The authors do not discuss runtime execution of models.

Lifting of Metamodels to Ontologies

In [43], the authors discuss how metamodels of modelling languages can semi-automatically be lifted to ontologies with the purpose of improving the integration of the metamodels. By lifting metamodels to ontologies, matching between metamodels can be performed at a conceptual level, where implicit domain concepts in the metamodels are made explicit (e.g. concepts represented by class attributes are extracted and instead represented as classes). Using ontologies improves the foundation for logical reasoning and instance classification which may be of support in the integration process. The lifting process comprises three steps. A metamodel is first converted into a *pseudo-ontology* (using *Atlas Transformation Language (ATL)*[10] transformations). Second, the pseudo-ontology is refactored to unfold hidden concepts which should be made explicit in the ontology. Lastly, the ontology may be semantically enriched with axioms and it may be related to other available ontologies. The authors propose that matching of ontologies can yield more concise mappings between concepts than creating mappings between metamodels directly. A mapping consists of references to a source and target element and a confidence rate (from zero to one). Mappings between ontologies can be generated automatically and later refined manually. The mappings may be used for deriving *bridgings* (weaving model) between the original metamodels, which in turn can be used to derive model transformations. A part of the motivation is that the ontology refactoring addresses heterogeneities with respect to how metamodels are defined, i.e. that semantically equal concepts may be modelled differently in metamodels.

Assisted Integration Using Text Analysis

The semantics of domain-specific modelling languages are typically expressed in natural language. The authors of [48] claim that metamodels do not contain enough semantic information to support integration decisions, i.e. finding what concepts of different metamodels that overlap semantically. In particular, if the terminologies used in the different metamodels are too far apart then additional information is required to make educated choices. Hence, the authors propose using analysis of informal semantics specifications to present concept candidates that are semantically similar. By evaluating the proposed candidates from the analysis with a manually specified reference mapping between concepts of two languages (two cases/language pairs were used), the authors conclude that the proposed candidates help finding semantically similar/matching concepts.

[10]http://eclipse.org/atl

3.1.6 Component-Based Integration

Components Comprising Code and Models

Integrating models by defining them as components is another branch of model composition. The authors of [47] present an approach where both code and executable models are grouped together in components. The functionality of a component is either implemented purely in code, in one or more models or as a combination of a code base and one or more models. The behavioural semantics of the models are defined in terms of interpreters. The code and models are connected via a mediator module. The mediator ensures that exchange of data and invocation of operations (both ways) are possible. A component has two types of ports: *functionality* ports and *management* ports. Functionality ports act as conventional component access points, whereas management ports allow accessing and manipulating the internal constituents of the component, e.g. querying and transforming the models. The ports are reified as Java interfaces and allow a component to be connected to other components or tools. A component can be adapted (referred to as tailoring) to create variants (e.g. different products in a product line) and adapted at runtime to address changes in the operating environment of the component. A reference implementation has been defined using Java and *OSGi*[11]. The authors motivate their approach by arguing that the alignment between code and models is improved when both these artefacts are grouped in components. Furthermore, inconsistencies are reduced by interpreting the models directly, i.e. no code is generated from the models.

Construction of Languages using Variability Modelling

Constructing languages from smaller building blocks/components is a viable approach. However, as argued by the authors of [49], this approach requires handling implicit dependencies between language assets in order to produce the languages. These dependencies typically need to be addressed manually by a language developer. The authors propose using variability modelling for expressing dependencies, which simplifies how languages are assembled and makes it easier for domain experts to participate. The approach utilises *Neverlang* [84] - a framework for building DSLs using BNF grammar [77], and the *Common Variability Language (CVL)*[12]. CVL is a language for management of variability for models conforming to MOF metamodels. Language components are defined in Neverlang. A component is a slice (Neverlang terminology) that refers to a piece of syntax and a role defining the semantics for the syntax. Syntax and roles are defined in modules. The construction of languages from the components is achieved by creating a variability model for a family of languages. Creating the variability model is supported by inspecting a dependency graph which maps the dependencies between the components. The graph

[11]http://www.osgi.org
[12]http://www.variabilitymodeling.org

can be derived automatically by analysing the components. Building a specific language is achieved by selecting a set of features as described by the variability model (resolving the variability). Composition directives are derived from the selected features (resolution model) by using a derivation operator. Neverlang then combines the components and produces an interpreter. The composition of the components is *syntax-driven* and *implicit*, i.e. a grammar rule defined in one component may depend on non-terminals defined in other components.

3.2 Model Migration and Co-evolution

Research on co-evolution can be traced back three decades to evolution of database schemas with co-evolution of data [113]. However, here we will only focus on recent approaches for model (artefact) co-evolution. We organise the available mechanisms according to whether they calculate metamodel differences or use a stepwise adaptation/co-adaptation. Also, we include some other approaches to model migration.

3.2.1 Difference Models

Automatic Co-evolution of Models

In [57], the authors discuss how model co-evolution can be performed automatically by calculating a difference model representing changes between two metamodels. The difference model is an instance of an extended *KM3 meta-metamodel* [85]. The extended metamodel contains constructs for modelling each individual change performed in a metamodel, e.g. the addition of a class or removal of a reference. A higher-order transformation generates a model transformation based on the difference model.

The authors categorise metamodel changes either as *parallel independent* or *parallel dependent*. Parallel independent changes may be performed in an arbitrary order since these do not relate each other. Parallel dependent changes require a specific order to achieve confluence, since the changes relate to each other. Resolution of dependencies is discussed further in [70].

The difference model is produced by an automatic analysis of the two metamodels (e.g. using *EMF Compare*[13]). It is then used as input to higher-order transformations in ATL which produce the model-to-model transformations (also in ATL) when executed. The model-to-model transformations co-evolve the models. Parallel dependent changes are briefly discussed. These are solved by calculating a *difference powermodel* which contains all possible submodels of a given difference model. The intention is that the dependent changes can be reduced to independent changes which may be handled automatically.

[13]http://www.eclipse.org/emf/compare

Co-evolution of Artefacts using EMFMigrate

EMFMigrate is an approach for co-evolving modelling artefacts in a uniform way [18]. It is based on specifying migration rules which can further be decomposed into rewriting rules. Each rewriting rule is evaluated with respect to a provided difference model (which is manually specified or automatically generated), and applied if the guard (boolean expression) of the rule is true. Several migration rules comprise a migration program. A migration program may import a library of migration rules pertinent to a particular migration scenario, e.g. a library with migration rules defined accordingly to the ATL metamodel or Ecore metamodel.

Adaptation by Detection of Changes

The work of [59] resembles the approach in [57] by how co-evolution is achieved by deriving and executing model-to-model transformations. The main difference is how the difference model, or matching model as it is named in [59], is computed. Specifically, the approach in [59] is based on using heuristics (comprising a matching strategy) to identify equivalences and differences between two metamodels. A part of the motivation for using heuristics is that complex changes, i.e. a set of actions that affect multiple metamodel concepts, can not be appropriately identified using tools such as EMF Compare. The authors claim that heuristics are needed because comparing metamodels is an NP complete problem. Each heuristic produces an intermediate matching model. The final matching model is used as input for a higher-order transformation that produces the model-to-model transformation.

Migration using a Graphical Representation of a Difference Model

An approach based on specifying the difference model using a graphical notation is discussed in [60]. Contrary to [57] and [59], the model is specified explicitly by the user. Migration rules are expressed using classes and relations as found in two versions of a metamodel (i.e. the old and evolved metamodels). The metamodel elements comprising the migration rules are mapped to indicate an evolutional step, e.g. a class named X in the old metamodel may be mapped to a class named Y in the evolved metamodel indicating that the X class has been given the new name Y. Additional conditions and commands (e.g. for querying models and setting attribute values) can be specified imperatively in C++ to describe more complex migration rules. Breaking and resolvable changes are automatically addressed, whereas breaking and unresolvable changes are dealt with manually. Code for migrating the models is generated automatically.

3.2.2 Stepwise Adaptation

Adaptation of Metamodels with Co-adaptation of Models

The author of [58] discusses an approach (theoretic framework) for simultaneous adaptation of metamodels and models. That is, models are co-adapted as a consequence of applying well-defined adaptations to their metamodel. Models are co-adapted by executing co-transformations. A co-transformation is created by instantiating a co-transformation pattern. Each metamodel adaptation instantiates a suitable co-transformation pattern. The metamodel adaptations are classified according to their semantics- and instance-preservation properties yielding three categories of adaptations: *refactoring*, *construction* and *destruction*. The provided framework also shows how properties concerned with semantics-preservation are correlated with properties of instance-preservation.

Coupled Evolution of Metamodels and Models

Changing a metamodel implies applying a derived change to the existing models. In [63], such a coupled change is known as a coupled transaction. Evolution and co-evolution of metamodels and models, respectively, can thus be expressed by composing any number of coupled transactions in a sequential manner. The authors discuss two types of coupled transactions: *reusable* and *custom*. Reusable coupled transactions represent metamodel-independent changes that are applicable to several different metamodels (using parameters), whereas custom coupled transactions are pertinent to specific metamodels. The authors claim that a criterion for a successful migration approach lies in the ability to reuse migration knowledge and at the same time allow for expressiveness. Reuse is addressed in terms of reusable transactions that can be picked from a library. Expressiveness is achieved by providing the user with a Turing-complete language containing primitives for adaptation and migration. The approach, referred to as *COPE*, has been realised on top of EMF; also known as the *Edapt* framework[14].

Operators for Coupled Evolution of Metamodels and Models

In [64], the authors discuss a set of 61 reusable coupled operators for evolution of metamodels with automatic co-evolution of models. The operators are classified according to three criteria: *language preservation*, *model preservation* and *bi-directionality*. The aim of the authors is *practical completeness*, i.e. a catalogue of operators that can be used to realise the most common evolutions of metamodels with co-evolution of models. The operators have been identified by studying the state of the art and as a result of performing case studies. The operators address evolution concerned with core metamodelling constructs, and do not support evolution regarding e.g. operations, derived features or constraints.

[14]http://www.eclipse.org/edapt

Migration of Models with Support for Multiplicity Changes

Changing the multiplicity of an association or adding an association between two classes in a metamodel may yield multiplicity constraint violations. This means that an existing model of the metamodel no longer fulfills the constraints implicitly specified by the multiplicities of the association, i.e. objects may need to be added or removed from the model. In [62], multiplicity constraint violations are addressed by a rule-based model migration approach. The approach first handles structure and type migration (merging of classes, moving properties, etc.) and then resolves multiplicity constraint violations. The authors discuss association inheritance and how this can be addressed. Model migration is given in the form of transformation rules, i.e. in the *Henshin* model transformation language[15]. Model migrations can be generated automatically. However, manual intervention and customisation of migrations may be required. The authors claim that their approach can result in correctly typed models with no violation of multiplicity constraints in a finite number of migration steps as long as the metamodel is *finite satisfiable*. A finite satisfiable metamodel means a metamodel whose association multiplicitiy constraints are carefully chosen such that there exists at least one model that fulfills them. Whether a metamodel is finite satisfiable can be decided by solving a linear system of inequalities.

3.2.3 Other Migration Approaches

Migration using Conservative Copy

A model transformation is either of type *new-target* or *existing-target* [66]. A new-target transformation initialises an empty target model and builds the model sequentially based on the contents of the source model. An existing-target transformation initialises the target model as an exact copy of the source model and deletes elements to reflect requirements of the transformation. In [66], a hybrid transformation type is discussed; known as *conservative copy*. Conservative copy means that the target model is initialised only with elements from the source model that conforms to the target metamodel. A user-controlled version of conservative copy is also discussed in which a type mapping function may be used to specify an evolved metamodel type (class) for each original metamodel type. In addition, it is possible to specify ignored features which are not automatically copied to the target model. Based on an analysis, the authors claim that using conservative copy in the general case requires using fewer model operations than using new-target or existing-target transformations. Hence, they believe conservative copy transformations are better suited for model migration. The authors have created a migration language named *Epsilon Flock*, and as the name suggests, it is built on top of *Eclipse Epsilon*[16] - a platform for realising model management. Flock is a rule-based transformation lan-

[15]http://www.eclipse.org/henshin
[16]http://www.eclipse.org/epsilon

guage supporting both declarative and imperative code. It uses conservative copy for migrating models. The authors state that Flock is best suited for migrations that do not encompass large-scale metamodel revisions.

Detecting Co-evolution Failures

In [61], the authors present an approach for detecting discrepancies between a metamodel and its models. This includes analysing models with respect to their metamodel after a third party co-adaptation mechanism has been applied. Specifically, models may not be co-adapted correctly using such a mechanism. The cornerstone in the approach is the use of constraints that enforce syntactic and semantic properties. The validity of a model with respect to a metamodel (conformance with additional constraints) may be proven by analysing the constraints. Constraints are updated as a result of applying changes to a metamodel. As pointed out, a model may not have been co-evolved correctly (or not been co-evolved at all). In this respect, the authors discuss how options for correct co-adaptation can be derived and suggested to the user by a reasoning engine (not yet implemented). This includes the ability to co-evolve models automatically in some situations. The authors claim that their approach may address situations with side effects, i.e. when an executed model adaptation may induce invalidity in other parts of a model.

The foundation for the work is discussed in [65] where the authors present an approach for co-evolving constraints as changes are applied to a metamodel. The approach is based on using constraint templates with variables, that are instantiated to generate or update constraints enforcing rules for elements in a metamodel. A constraint template may e.g. address reference multiplicities and hence be instantiated multiple times to produce constraints than enforce the multiplicities for each reference in a metamodel. Management of constraints is performed automatically by a template engine in an incremental manner, i.e. focusing on only the changed metamodel elements.

Model Migration using Attributed Graph Transformations

A transformation language and environment, named *Henshin*, for in-place transformations of EMF models are discussed in [67]. An *in-place* transformation operates directly on the (source) model, whereas an *out-place* transformation creates a copy of the source model and, hence, does not change the source model. Henshin allows specifying transformations using a declarative style, which allows formal reasoning about the transformations. It can be used to specify model migration using manual specification, i.e. transformation rules are specified for both a metamodel and its models. A rule is expressed as two graphs in a visual syntax; a left-hand side graph describes a pattern identifying where the changes specified by a right-hand side graph should be applied. (The two graphs may be combined into one graph.) The authors state that Henshin can potentially be used for an operator-based approach as well.

Co-evolution with Transformation of Wrapped Models

In [69], the authors present an approach for evolution and co-evolution of metamodels and models, respectively, using graph transformations as specified in Henshin. The underlying idea of the approach is that transformations for both the metamodel level and the model level can be expressed in the same graph(s). This is achieved using model object wrappers.

In EMF, all model objects are instances of classes indirectly implementing the EObject interface. The type of a model object, i.e. the metamodel class of which a model object is an instance, can only be determined by using reflection. The approach addresses this by wrapping each EObject instance in a wrapper object (instance of WObject) with the purpose of making type-instance relations between a metamodel and model available as structural features. This way, the type of a model object (EClass object), the type of a link instance (EReference object) and the type of a value instance (EAttribute object) can be accessed using references instead of reflection. Also, object values and links can be accessed in a generic way.

By specifying metamodel and model level changes in the same graph it is possible to increase the expressiveness of migration rules (supporting metamodel-specific changes). That is, the metamodel changes and the corresponding model changes are represented directly at the same conceptual level without the need to sequence a set of low-level co-evolution operations. Metamodel evolution and model migration are achieved by transforming the metamodel and an existing model according to operations specified in the graph (e.g. preserve, create, delete). The evolved metamodel is directly available in the graph, whereas a co-evolved model is acquired by extracting the changed EObject instances from the wrapper objects. Hence, evolution of the metamodel and a model is performed as an atomic step (the approach also supports delaying the execution of the model migration).

A Formal Approach to Metamodel Evolution

A technique for model migration using diagrammatic specifications in terms of graph theory and category theory is discussed in [68]. Metamodel modifications are specified using metamodel evolution rules (transformations), whereas models are transformed by migration rules corresponding to the rules applied on the metamodel. Specifically, the left and right hand sides of a migration rule conform to the left and right hand sides of a metamodel evolution rule, respectively.

3.3 Model Templates and Model Typing

In this section we discuss approaches for model templates and model typing.

3.3.1 Model Templates

Genericity by Means of Concepts

The authors of [42] argue that reuse, genericity, modularity and extensibility in MDE artefacts can be improved by utilising requirement-centric specifications instead of type-centric specifications. Specifically, they propose using *structural concepts* for specifying structural and behavioural requirements. A concept adds a level of indirection and allows defining behaviour in a more abstract way, instead of directly for a particular metamodel. The behaviour of a concept may be applied non-intrusively to all metamodels that fulfill the requirements of the concept. The notion of *concept* refers to the set of requirements required by type parameters in generic programming to ensure correct template instantiation and execution. A concept is a pattern specification of structure that needs to be found in a metamodel (for the concept to be applicable to the metamodel). Hence, the concept is defined in terms of a metamodel. The elements of the metamodel comprising the concept definition act as variables and are bound to elements of a specific metamodel(s). Example applications for concepts are generic simulators and code generators. These may be executed for all instances of metamodels that are compatible with the concepts used to define the simulators or generators.

A concept can only be bound to metamodels that fulfill the structural requirements of the concept. To increase the flexibility of concepts, the authors propose *hybrid concepts*. A hybrid concept specifies structural requirements in terms of operations that the specific metamodels need to implement. Using operations allow for greater variations in the metamodels. A structural concept may *realise* a hybrid concept with a default implementation of the operations in the hybrid concept. Hence, a metamodel may be bound to a structural concept that provides an implementation of the operations instead of binding directly to the hybrid concept. Similarly, a concept may also be *specialised* to create hierarchies of concepts.

An approach for defining reusable models as *model templates* is also discussed. A model template contains a model (fragment) and uses a concept to specify interface requirements. The concept is defined by using types from a metamodel, according to a *typed on* relation. Models conforming to this metamodel are valid actual parameters in instantiations of the model template. Hence, the model (fragment) of the template may be composed with models used as actual parameters without modifying the metamodel of the models. A model template can utilise several concepts for expressing more than one interface. This means that several models (used as actual parameters) may be combined using a model template. A derivative of model templates is *generic model templates*. A generic model template defines its interface requirements in terms of a concept that can be bound to different metamodels, instead of a concept that is typed on one particular metamodel. Hence, the variables of the concept may be bound to elements of different metamodels. Generic model templates therefore express patterns that can be applied to families of metamodels. A generic model template may be instantiated with models conforming to all compatible metamodels, i.e. metamodels that fulfill the requirements of the template concept.

Finally, the authors discuss *semantic mixin layers* in the form of metamodel templates. A metamodel template supports extending a metamodel with structure for realising behaviour. The behaviour is defined externally (e.g. as a simulator) over the generic types of the template and an associated concept. All metamodels that fulfill the requirements of the concept may be extended with the associated behaviour of the template. The metamodels are extended using a mechanism similar to package merge. Hence, an instance of a template is an extended version of a specific metamodel. Only structure that is optionally with regard to model instantiation may be added to the metamodel to ensure conformance between existing models and the extended metamodel[17]. One type of application for metamodel templates is to add a simulation infrastructure to metamodels. The authors state that concepts can be thought of as representatives for (meta)model types.

3.3.2 Model Typing

Model Type as a Collection of Object Types

In [53], the authors discuss how a model type can be defined as an extension to object-oriented typing, i.e. as a collection of model object types (a set of metaclasses) and their relations. The approach is centred around a model-type conformance relation which indicates whether two model types are type compatible. The conformance relation is based on type matching and states that a model type matches a reference model type if the model type contains equally named types/-metaclasses as the reference model type where the respective classes have the same number of properties (e.g. relations) and operations whose signatures are identical. The conformance relation allows building tools/model-related services that have an input parameter typed by a required model type. Models that are type compatible with the required model type may be read and processed by the tools/services. That is, models that are typed by a provided model type which in turn is conformant with the required model type of the tools/services. Model types can be used in expressions and to type variables. The conformance relation also dictates whether a model type may be substituted with another. Model types allow tools and other artefacts like, e.g. transformations, to be reused in a flexible manner.

[17]Not discussed in the paper, but clarified with author.

Metamodel Refactoring using Model Types

The authors of [118] present an approach for defining generic transformations by means of model types, aspect weaving and derived properties. The approach works by defining a generic metamodel with representations of common concepts (classes) that are found in a set of target metamodels; metamodels for which the generic transformations should be valid. The target metamodels are adapted by weaving in aspects, i.e. additional properties and derived properties are added with the purpose of making the target metamodels type compatible with the generic metamodel. Transformations, in this case refactoring transformations, can then be written once according to the generic metamodel and then be applied for all the target metamodels. Model typing supports establishing a matching relation between the classes (types) of the generic metamodel and the classes of the target metamodels, respectively. This in turn supports substitutability when it comes to applying the transformations. In this work two constraints regarding matching of classes are removed. First, the classes can have different names as long as they are structural compliant, i.e. as discussed in [53]. Second, the matching also applies to subclasses of a given class.

3.3.3 Subtyping of Model Types

Substitutability is an important property in object-oriented approaches. In [54], the authors present four subtyping relations for applications on models. In particular, the discussion relates to how model types may be subtyped, which allows subtyping at the model level. The key criterion for a subtyping relation is that a model typed by one model type can be safely substituted by a model typed with another model type. This in turn means that model manipulations, e.g. transformations, can be reused. The authors discuss how adaptations may be applied to induce a valid subtyping relation (making the structures of two model types isomorphic, i.e. equivalent). They also discuss partial model substitutability where only a subset of a model type, known as the effective model type, is considered.

Chapter 4

Research Method

The research method used to structure and evaluate the work of this thesis is known as *technology research* [1]. It closely resembles the classical research method and comprises three main phases: *problem analysis*, *innovation* and *evaluation*. In classical research the problem is the need for new theories that explain the world around us. Technology research is about creating artefacts that improve properties of systems or processes, and is therefore often *applied research*. On the contrary, classical research is typically *basic research*.

Technology research is an iterative process revolving around the three identified phases. The initial step of the process is the identification of a problem that can be understood as a need for a new or improved artefact. The result of the problem analysis phase is a set of requirements that the artefact must fulfill. In the innovation step, an artefact is constructed according to the identified requirements. The overall research hypothesis is that the artefact satisfies the identified need as specified by the requirements. The overall hypothesis is decomposed into sub-hypotheses which assert desirable properties or qualities of the artefact. The hypotheses are tested in the evaluation phase by constructing predictions. The requirements are the basis from which predictions are made. It can then be argued that an artefact fulfills its need by generalising from a set of verified predictions (which also means that the requirements are satisfied). Evaluation may lead to new insight that is further used as input for a new problem analysis phase.

Three types of scientific methods in computer science are differentiated in [107]: *theoretical computer science*, *experimental computer science* and *computer simulation*. Conceptualising and designing artefacts with evaluation in terms of prototypes, as discussed in this thesis, belongs to the discipline of experimental computer science. We have also worked according to what is considered theoretical computer science, e.g. in terms of formalising desirable properties of an artefact.

In the context of this thesis, the artefacts are in the form of language constructs, frameworks and theories. The artefacts realise a number of mechanisms. The requirements for the artefacts are given in Chapter 5.

4.1 From Problem Analysis to Artefacts

The problems investigated in this thesis were identified by studying the state of the art. From the start of the project we chose to have a broad view on model-driven engineering and domain-specific modelling with the purpose of identifying weaknesses of available mechanisms for metamodelling, in particular model composition and adaptation. We identified several existing modelling mechanisms that would be relevant to evaluate with respect to metamodels, e.g.: package templates, class nesting and generic types. We first investigated using generic types in relation with class nesting, and later moved on to study whether (and how) package templates could be applied for metamodels. Even though several approaches for (meta)model composition exist, we could not find one that supports composition of metamodels that have operational semantics defined by class operations. In other words, we could not find related work on metamodel composition approaches that are type-safe (with the exception of aspect-oriented programming). During the course of the project we shifted focus to address model co-evolution as this is critical when metamodels are adapted. Approaches for model co-evolution, i.e. model migration, are typically realised as either matching-based mechanisms or as mechanisms that adapt a metamodel and its models in a stepwise coupled manner. We defined an analysis framework for composition and adaptation of metamodels with co-evolution of models, that works as an operator-based approach with change recording. Composition of metamodels can be seen as a form of evolution. However, we did not find related work on migration approaches that explicitly treat metamodel composition as an evolution, and support generating transformations for either migrating or composing existing models (two choices). Working on model migration inspired us to seek solutions to how models may be combined without compromising the validity of related model artefacts like tools and editors. This included elaborating the theory on how class nesting can be used to define a typing scheme for metamodels. The problem analysis is thoroughly discussed in Chapter 5.

4.2 Innovation

The innovation pertinent to the work of this thesis is the design and realisation of the artefacts. For Artefact A, the innovation lies in how the package template concept has been elaborated to be useful for metamodels. The innovation regarding Artefact B is how transformations for model migration can be calculated based on instantiation directives. This includes generating transformations reflecting how metamodels evolve throughout a template hierarchy. Artefact C provides a new way of analysing and verifying metamodel composition and adaptation. A part of the innovation is how the analysis has been formalised. Also, both Artefact B and Artefact C support composition of models following metamodel composition. For Artefact D we have invented a new way for integrating the operational semantics of metamodels. The innovation is how integration is specified by means of mappings between metamodel elements and how proxy classes can be used as substitutes for other classes. Artefact E is based on using class nesting for defining metamodel types. The innovation is here a theory that explains how class nesting can be applied for typing of metamodels. The artefacts are discussed in Chapter 6 and in Chapter 7.

4.3 Evaluation

We have implemented four prototype tools/frameworks that have been used to evaluate the artefacts. We have used the tools/frameworks on several example cases (which can be seen as simple case studies [116]). Adaptation and migration of Petri net metamodels/models is a well-known problem in the literature on model migration approaches [75]. We therefore chose to evaluate Artefact A, Artefact B and Artefact C on Petri nets. That is, adaptation of Petri net metamodels (including model migration) that do not require deleting elements or changing the multiplicities of references (i.e. from the metamodel μ_2 to the metamodel μ_5 as given in [58] including intermediate steps)[1]. We have also partially validated Artefact A and Artefact C by adapting the GMF Graph metamodel [102]. For Artefact D we used a non-trivial problem where the semantics of a GPL is integrated with the semantics of a state machine language. We have also used other examples to test the artefacts. Artefact C has additionally been evaluated analytically in terms of proving that re-establishment of model conformance during model migration can be guaranteed. For Artefact E we have used logical argumentation for evaluating the use of class nesting for defining metamodel types. The prototypes are described in Chapter 6.

[1]The value for the `weight` attribute in the models has to be manually set to `1` since there is no support by the artefacts for setting specific default values for properties.

Chapter 5

Problem Analysis

In this chapter we present the problems that are addressed in this thesis and motivate their importance. We first briefly present the current challenges and goals in MDE as these are the origin of the specific problems we have solved. We then concretise further according to the three areas of study, i.e. metamodel composition, model migration and model typing, in the form of a problem statement. A list of requirements for the artefacts and overall research questions are then given. The requirements are later used in the evaluation of our work.

5.1 Challenges and Goals in MDE

MDE is a promising approach for addressing the increasing complexity in software development and for improving productivity. However, there are several challenges and open questions that need to be addressed. The challenges are related. However, for clarity we divide them into six categories.

5.1.1 Complexity and Ease of Use

One of the key challenges in software engineering is the increasing complexity both with regard to the problems that need to be solved and the software that need to be created to solve these problems. This complexity needs to be addressed in structured and intuitive ways. In other words, the complexity must be handled by decomposing problems and software into smaller concerns which are easier to understand and reason about. One goal such wise is to create software that is easy to use and helps understanding the problem at hand. Specifically, complexity should not be solved by introducing new complexity.

Tools are an important part of MDE, e.g. tools for model management. Unfortunately, the tools available are not typically perceived as easy to use [89], e.g. language development tools which are not necessarily designed with the end user in mind [49][56]. The immediate drawback of this is that the technological threshold and learning curve for using MDE are significant [16]. MDE concepts are also difficult to understand, e.g. because of a high level of abstraction [17]. These issues in turn reduce the chances of acceptance and adoption of MDE in the industry [89]. More focus has to be put on creating solutions for involving all stakeholders; including those that do not have the technical expertise of software developers. Enabling communication and interaction between technical and non-technical experts is considered as one of the successes of MDE [17]. Hence, MDE should help by reducing complexity and not making the software engineering process itself too complex [17].

5.1.2 Costs and Productivity

Improving productivity is a challenge in all business endeavours. In software engineering, productivity depends on several things including the level of complexity the engineers must handle, the type of problems at hand and the quality of available tools and mechanisms. Also, the degree of automation that can be used directly impacts how much time that is spent on a given problem.

Productivity may be assessed in terms of short-term productivity and long-term productivity. Short-term productivity says how much functionality a certain artefact may currently deliver, whereas long-term productivity depends on how resilient an artefact is to change and how valuable it is in the long-term, i.e. an artefact's longevity [10]. Improving productivity can indirectly be seen as the effort of increasing an artefact's current and long-term value. In [10] it is argued that increasing an artefact's long term value is strategically more important than primarily focusing on an artefact's current value since the return on investment will be greater. That is, reuse of solutions is a focus point.

To increase productivity in MDE it is necessary to improve how artefacts, e.g. models, metamodels and transformations, can be managed, evolved and reused [89]. The overall goal is to increase the artefacts' long-term value. Language development is a central activity in MDE which is both complex and time-consuming. Languages are often made from scratch [54] even though reuse would be possible by applying the right techniques, e.g. by composing already available modules [49]. Reusability increases productivity and reduces costs, and has been recognised as a key issue in modelling language design [16]. However, composing modules is typically only feasible for skilled language developers [49] which, as mentioned, increases the threshold for adoption of MDE in industry. There is also a cost and risk of applying MDE because of required changes to the software development process [89].

To increase the long-term value of a domain-specific language and hence make it economically viable, the language should support modelling of many applications [36]. One way of achieving this is to support adaptable metamodels that can be tailored for different domains [61]. Finding optimal solutions to this is an open question [16]. Moreover, design tools should be adaptable as well for supporting a variety of disciplines, domains and applications [65].

5.1.3 Evolution and Traceability

Software evolves and requires maintenance. There are many types of evolution including e.g. architecture evolution and language evolution [82]. Evolution can also be categorised according to the types of changes applied, e.g. if the evolution is a consequence of improving the longevity of software by addressing new requirements or whether the evolution results from correcting errors in the software. (Support for scalability is also an important goal.) Hence, software changes continuously which consumes a large amount of resources [82].

Traceability allows tracking changes in software and supports e.g. impact analysis and verification of requirements. In particular, impact analysis helps determining the consequences of applying changes. Supporting traceability is therefore desirable.

The need for change drives evolution of MDE artefacts [82]. Evolution can be seen as the process of re-establishing consistency between an artefact and its problem domain, and is one of the main challenges in MDE [15][17]. New or changed requirements is one of the fundamental drivers for change in software engineering [10]. In MDE this is particularly evident because of the focus on using domain-specific modelling. Domain-specificity comes at the cost of being highly sensible to change [71]. A DSL has a constrained number of problem-specific constructs which allow modelling a certain type of problems. Or put differently, only a pre-defined set of problems can be modelled as opposed to using GPLs in which practically any kind of problem may be expressed and solved. The constructs of a DSL are typically rigidly defined and there is little room for supporting variance regarding the types of problems that can be modelled or the accuracy with which the problems can be modelled. Consequently, desirable changes have to be reflected by evolving the languages; meaning that the language constructs have to be adapted to meet new requirements or the language have to be extended with new constructs. This includes adapting language notation, e.g. concrete syntax and model editors.

Evolution of an artefact can typically not be seen in isolation from other artefacts in the metamodelling ecosystem. The reason is that artefacts are related and changing one artefact may require co-evolving other artefacts to maintain consistency [15][3]. Assessing the impacts of changing a model or metamodel has been identified as a major challenge in model evolution [16]. Specifically, changing metamodels normally impacts several other artefacts as these are defined relatively to the metamodel.

5.1.4 Verification and Correctness

The most important concern regarding software is that it performs correctly. This requires rigorous testing of software and verification of its properties. And important concept in this regard is *determinism*, i.e. the ability to derive the future state and operation of software based on currently available information.

Verification and validation are important activities in MDE and there has been a lot of work on formal approaches for supporting MDE [16]. These activities require a formal foundation. Two challenges are to integrate tools for formal verification into modelling environments and defining DSLs that enforce model correctness [16].

5.1.5 Type-Safety

Type-safety is an important property of software which ensures that type errors do not occur during execution of code. In MDE, type-safety should be verifiable prior to execution of the operational semantics. For metamodels that specify operational semantics, type-safety is checked by the type-checker for the action language used to specify the code. Second, model management operations and mechanisms (e.g. model typing) that affect the operational semantics should seek to preserve type-safety. For example, when both the structure and operational semantics of metamodels are composed and/or adapted, it should be possible to verify that the execution of the resulting operational semantics does not result in type errors. We will discuss a couple of scenarios of composition and adaptation of metamodels that will induce type errors if they are not addressed explicitly.

Renaming classes is a commonly used basic operation in MDE. Classes are used to type variables, references, operations and operation parameters. When a class is given a new name, a type error is introduced for all elements that are typed with the class. To rectify this, all declarations that use the old class name have to be updated to use the new name of the class. The same correction has to be done when classes are merged since at most one of the source classes may retain its name.

Renaming of classes also induces type errors within code (operational semantics). An example of this is an expression where a class is being instantiated. When the class is renamed as an adaptation, a type error is induced because the code tries to instantiate a class that is no longer accessible using the old name. Type errors may also occur if a class is given a new name whereas the old name is used for another class. The result of this is that code referring to the old class name will now unintentionally be referring to the other class which has taken the name. Clearly, this class is not type-compatible with the old class.

In other words, code that was type-safe prior to the composition/adaptation has to be type-safe after the composition/adaptation. This also implies that e.g. all features and operations accessed on an object before composition and adaptation are available on the respective object after composition and adaptation. This typically means that features and operations can not be deleted.

5.1.6 Representation and Modularisation

A problem may typically be modelled and solved in many different ways. Finding an optimal solution includes creating an optimal representation of the problem. This includes selecting an appropriate abstraction level and deciding how to segment the problem in views/viewpoints (concerns) and perspectives. This includes multi-view software that supports participation of different stakeholders. Important aspects are reusing information effectively and avoiding replication of information.

Using multiple DSLs to realise a design is a common approach in MDE [56]. This allows addressing complexity and hiding details that are not relevant for a given concern [89]. A model may describe a single viewpoint. Viewpoints are typically dependent which requires defining relationships (correspondences) between elements of different viewpoints [90], which is not a trivial problem. The problem of relating different viewpoints can be differentiated into two subproblems: the conceptual integration of viewpoints and verification of consistency between the viewpoints [94]. Composing models representing different viewpoints and concerns is one approach for integrating the viewpoints in order to generate a global specification for a system.

5.1.7 Technological Heterogeneity and Compatibility

Allowing heterogeneous software and data to work in concert is another challenge in software engineering. This includes creating bridges between different technologies and integration with legacy software. Compatibility between different versions and variations of software is also a current challenge.

Using standardised metamodelling frameworks and meta-metamodels help alleviate difficulties regarding interoperability of technologies and tools. However, there are still many challenges in integrating different technologies while at the same time addressing challenges pertinent to evolution. Different technologies and tools do not evolve in a synchronised manner, which complicates the picture. On top of this, user-friendliness should be ensured. Heterogeneity is an increasing concern, e.g. because of how systems are built from several systems [17].

5.2 Problem Statement

In Chapter 2 we discussed model and metamodel composition, model migration and model typing, and the importance of these activities in MDE. In Chapter 3 we presented the available approaches and mechanisms for realising these activities. However, as discussed, there are still challenges and open questions that need to be addressed. In this section, we discuss the problems we have studied in further details and motivate their importance.

5.2.1 Metamodel Composition

Metamodelling is used increasingly in the industry. However, many approaches and tools available lack sufficient support for reusability, extensibility, modularity and ease of composition of artefacts, e.g. models and metamodels [42]. Model composition and metamodel composition are central operations in MDE that address these requirements. Improving model and metamodel composition therefore has a major impact on the usability of MDE and its applicability and adoption in industry. Model composition has been studied extensively. On the other hand, there are still open questions regarding metamodel composition that need to be addressed. Specifically, available tools have to improve in order to support evolution of metamodels and not only models [17]. This includes support for metamodel composition.

MDE is a model-centric approach, which requires the use of multiple domain-specific languages for creating the different models [56]. With respect to traditional software engineering, MDE adds two major new tasks to the software engineering process: language design and language integration. This implies that software developers need to master language design and their integration as well as using the languages for creating solutions. This requires tools that are both intuitive and simple to understand and use [89]. OMG issued a request for proposal known as the Metamodel Extension Facility RFP[1] in 2011. With the RFP, OMG solicits approaches that may replace UML Profiles for extending and integrating metamodels and subsets of metamodels. The request is still pending.

A model is formalised by a metamodel. Composing models therefore requires the metamodels to be composed as well to be able to acquire a formalisation of the composed model. In this thesis we focus on composition of metamodels defined by class models.

Correctness, Conflict Handling and Composition of Operational Semantics

One of the major limitations of current metamodel composition approaches is the limited support for ensuring and asserting correctness of the composition. In a large survey on aspect-oriented modelling approaches (for use in MDE) it was concluded that more sophisticated conflict detection and resolution mechanisms are needed [92]. Creating DSLs that enforce model correctness by construction has been identified as a challenge [16]. This challenge also pertains to operations on models and metamodels. Type-safety is a necessity when metamodels also define operational semantics. EMF and Kermeta are metamodelling frameworks that support defining the operational semantics of metamodels (languages) in terms of class operations. That is, the classes in EMF and Kermeta metamodels define types, with dynamic semantics, whose use can be type-checked[2]. With the exception of static introduction (open classes) in Kermeta [56], we are not aware of any dedicated mechanism that supports type-safe composition of metamodels with operational semantics defined in class operations.

In MDE, the dynamic semantics of a metamodel (language) can be defined in several different ways, including approaches for translational, operational, extensional and denotational semantics [2]. Most existing approaches in MDE (with e.g.

[1]http://www.omg.org/cgi-bin/doc.cgi?ad/2011-6-22 (accessed early 2017)

[2]EMF uses Java to specify the semantics whereas Kermeta uses an action language.

the exception of EMF and Kermeta) rely on using a translational approach where structural models are transformed to executable code. A disadvantage with this approach is that knowledge is kept both in the models and the code [46]. Changing either artefact requires updating the other dependent artefact. This results in information redundancy which may result in model and code not being aligned and thereby causing inconsistencies [47]. It may also make evolution more challenging and even lead to information erosion. It is argued that well-defined semantics at the model level is necessary for increasing the value of models [92]. Having this semantics defined within metamodels ensures that model and code appear as integrated logical units of functionality. Hence, there is a clear motivation for supporting metamodels that specify operational semantics, and composition of such metamodels. How can metamodels that specify operational semantics be composed type-safely without introducing a new layer of complexity? Can the operational semantics be composed simultaneously as the metamodel structure is composed? And, how can conflicts that appear be addressed in the operational semantics without having to manually changing the code?

Composition of Multiple Metamodels

Many of the available composition approaches support composition of only two (meta)models simultaneously. The approaches need to be used iteratively when more than two (meta)models need to be composed. Each iteration requires identifying what concepts to compose and resolution of conflicts (i.e. refactoring of the metamodels). Unfortunately, since each iteration only considers two metamodels, new conflicts may be induced with respect to further iterations. These conflicts later have to be addressed in the remaining iterations. Composing several metamodels iteratively may therefore result in unnecessary difficulties and reduce the usability of the composition approach. It can also be necessary to provide an ordering of how the metamodels should be composed which is not a trivial task. MDE suggests using multiple languages in concert [56], which motivates the importance of supporting composition of several metamodels simultaneously.

Reusability of Metamodels

Reusability has been recognised as a key issue when designing modelling languages [16]. Building (meta)models by selecting pre-made fragments is a solution for increasing productivity and reducing costs [42][49]. A goal is to promote reusable metamodel variants and at the same time avoid redundancy of variations. To increase applicability of the fragments it should be possible to adapt them for different domains [61]. Of this follows that a mechanism for composition of metamodels should provide means for adapting the metamodels as part of the composition. How can adaptations that preserve type-safety be supported? Is it possible to organise metamodel variations and versions in an intuitive way that avoids redundancy?

Conceptual Composition

One of the main difficulties with composing (and adapting) metamodels is the many artefacts in the metamodelling ecosystem that may be impacted by such an operation. There are two main types of solutions to this problem. First, the impacted artefacts can be updated to ensure compatibility with the composed (or adapted) metamodel. Second, instead of explicitly merging the structures of two (or more) metamodels during composition, semantic links are instead created between elements in the different metamodels. The links collectively represent a conceptual composition. The best known work on this is the Atlas Model Weaver (AMW) [22], which supports defining several types of links (correspondences) between elements in different models. As far as we know, there is no approach available that supports composing metamodels, that define operational semantics in class operations, using semantic links while at the same time integrating the operational semantics.

Integrating different languages by explicitly composing the structures of their metamodels is not trivial [43], it requires artefacts to evolve which is challenging [17] and it may even not be possible to acquire a composed metamodel for all involved languages, e.g. because the concepts of the metamodels reside on different abstraction levels and/or have differences with regard to semantics [90]. Is it possible to support type-safe integration of metamodels that specify operational semantics by means of semantic links? And, can such mechanism be completely non-intrusive to avoid inducing impacts on other artefacts?

Ease of Use

Composition of (meta)models is one of the key operations in MDE. Hence, its application should be as straightforward and efficient as possible, which is also stated in [16]. However, the available composition approaches are not straightforward to use and require skills in language development. Being able to have non-technical stakeholders participate more directly in the software engineering process is a goal of MDE. Furthermore, many mechanisms specify composition directives in a separate resource, e.g. a composition or weaving model, which results in additional artefacts that need to be maintained.

5.2.2 Model Migration

Metamodels formalise domain knowledge and are used as reference for a large number of artefacts [18]. The definition of these artefacts depends on the metamodel's definition. A metamodel evolves over time [61], which means that dependent artefacts (models, transformations and tools) are rendered invalid with regard to the evolved metamodel. One such artefact is models. Existing models will in many cases not conform to the resulting metamodel following an adaptation. This requires creating model-to-model transformations that update or migrate existing models each time a metamodel is adapted to ensure they conform to the evolved metamodel. This is a research challenge [91].

Transformation Required After Composition

Composing metamodels creates a situation where existing models of the metamodels may often not conform to the composed metamodel. There are two solutions to this. First, a model composition approach can be used for manually selecting two (or more) models which are then composed for acquiring a composed model conforming to the composed metamodel. Second, transformations can be made in order to update the models so that they conform to the composed metamodel. That is, the models are changed to meet the updated set of well-formedness rules imposed by the composed metamodel. The type of approach chosen for composing/updating the models depends on the premise for using metamodel composition in the first place.

For composition of metamodels defining languages for different viewpoints of a system it would typically be required to use a model composition approach for acquiring an integrated view of the system (utilising existing models). In situations where metamodel composition is used to e.g. increase the expressiveness of a metamodel, by composing a metamodel with some kind of additional features, it may instead be preferable to just update all the existing models of the metamodel with the minimum required objects/values in order to establish conformance with the extended metamodel. This is required when there is no models available conforming to the extension (metamodel fragment). This latter case of metamodel composition can be seen as an evolution since the composition can be described in terms of a number of sequential evolutionary steps, e.g. adding new classes and relations, and adding properties to existing classes (reflecting merging of classes). Hence, a migration approach that calculates a difference model from two metamodels may detect the result of a metamodel composition. Existing models can then be transformed to conformant versions of the evolved metamodel by generating default objects and values. However, there is no difference-based migration approach that supports model composition as a consequence of metamodel composition. Also, a difference-based approach can not be used when more than two metamodels are composed by only comparing the original and evolved metamodel.

Similarly, an operator-based migration approach may describe metamodel composition in terms of multiple operations. However, we are not aware of any operator-based approach that supports composing metamodels directly, with corresponding model composition. Again, this also requires the approach to support operating on several metamodels. Edapt[3] (based on COPE [63]) is an operator-based approach that supports an operation known as *Replace Class* which allows migrating instances of one class to another. This operation may in principle be used to imitate model composition. However, the use of the operation is constrained.

[3]http://www.eclipse.org/edapt

Migration approaches that rely on manual specification may address metamodel composition with corresponding model migration/composition, e.g. Flock [66], EMFMigrate [18] or Ecore2Ecore ([75]). However, there is an incentive for using automatic approaches. Supporting efficient migration of models as a consequence of changing metamodels is identified as a prerequisite for cost-effective MDE [71]. This includes addressing changes resulting from metamodel composition. Is it possible to define a model migration mechanism that also supports model composition straightforwardly as a consequence of metamodel composition?

Correctness

Differencing approaches do not guarantee producing a correct migration strategy with respect to the actual changes that were performed on a metamodel. For some domains, differencing approaches are therefore not suitable, and approaches that are deterministic and produce correct results are required [76]. Static analysis has been identified as a way of verifying conformance preservation in a model-independent manner [63]. Can analysis be used to assert that model migration is performed correctly?

5.2.3 Model Typing

Typing is one of the most successful concepts in programming languages. It allows specifying collections of entities that share a set of properties. The entities may be treated equally with respect to operations, and correctness may be asserted by means of type checking.

The metamodel plays an essential role in a metamodelling ecosystem as it represents a specification that many artefacts depend on. In other words, the artefacts are defined according to a specific metamodel which may be understood to have a specific type comprising its properties [42]. Typing is a concept that may alleviate the difficulties in supporting variance in the metamodel (due to required co-evolution). Some work has been performed on model types, e.g. [53]. However, the question on how to support metamodel types is still an open question.

Variance and Reuse

There are several advantages of supporting metamodel types. First, artefacts may be defined according to the properties of a metamodel type instead of the properties of a metamodel, which would increase the reuse value of the artefacts and ensure longevity [42]. A metamodel type constitutes an abstraction over several metamodels sharing a common set of properties and thereby allows reuse. That is, different metamodels may share the same artefacts and the artefacts are more resilient to metamodel evolution. This further means that a metamodel type represents the minimal set of metamodel properties that an artefact is required to handle.

Changing properties of a metamodel irrelevant to its type therefore does not (necessarily) impact the dependent artefacts. Second, types provide a means for realising polymorphism. This includes supporting genericity which allows specifying generic behaviour for metamodels [42]. Can metamodels be typed in a conventional way in order to utilise existing mechanisms that are based on typing? Is it possible to type metamodels and at the same time utilise type parameters? And, can several metamodel types be used in unison?

5.3 Requirements

Based on the problem statement, we present a list of specific requirements that capture desirable properties and characteristics of the artefacts discussed in this thesis. We also give a list of general requirements that are not tied to the specific problems we have investigated.

5.3.1 Specific Requirements

1. *Metamodel composition mechanisms should automatically ensure and assert the correctness of their application to the extent possible.* Having metamodel composition mechanisms that produce correct results is essential for creating high quality MDE solutions. It means that manual verification and correction is reduced to a minimum. This increases confidence in the software and improves productivity, which in turn is important for gaining acceptance of MDE.

2. *Composition of operational semantics should be supported for metamodels that define such in class operations.* Supporting composition of metamodels' structure *and* operational semantics ensures that additional mechanisms do not have to be applied for achieving an integrated semantics for the composed metamodel.

3. *It should be possible to statically type-check metamodel compositions for metamodels that specify operational semantics in class operations.* Type checking is essential when composing operational semantics. It ensures that the execution of models conforming to the composed metamodel does not result in type errors, i.e. the integrity of the composed semantics is high. The type checking should be static. Dynamic type checking is not desirable since this may require recomposing the metamodels if type errors occur at runtime. The added flexibility of dynamic type checking is not needed when composing metamodels.

4. *A metamodel composition mechanism should support composing an arbitrary number of metamodels simultaneously.* Being able to consider several metamodels simultaneously during composition improves conflict handling and increases

agility. It also means that there is no need to order the metamodels prior to composition. To achieve this, the mechanism needs to support symmetric composition.

5. *Metamodels should be organised in a way that eases reuse and avoids redundancy of metamodel variations.* Organising metamodels according to different variants and versions improves management of evolution. It makes it easier to identify the most suitable metamodel variant for a given task and ensures traceability by supporting the generation of a change history.

6. *Approaches for composing metamodels should take artefacts that are dependent on the metamodels into consideration; in particular the models.* Co-evolution is typically required when metamodels evolve. Having means for co-evolving artefacts or making artefacts resilient to metamodel changes improves how evolution is handled.

7. *Approaches for metamodel composition should support adapting metamodels as part of the composition operation.* It may be necessary to adapt metamodels to facilitate composition. Supporting such natively by the composition mechanism reduces the need of adapting metamodels prior to composition.

8. *Mechanisms supporting adaptation of metamodels should support both metamodel-specific and metamodel-independent adaptations (coupled changes).* Supporting metamodel-specific changes are important to reflect particular problem domains, whereas metamodel-independent changes allow reusing adaptations (and migrations) for several metamodels (and models).

9. *Migration approaches should produce conformant models automatically, to the extent possible, and verify correctness of the migration.* By producing conformant models, manual intervention is reduced to a minimum. In particular, this is advantageous when composition is used to extend a metamodel with new features.

10. *It should be possible to type metamodels.* Having a clear notion of a metamodel type improves how variance and reuse can be addressed in the metamodelling ecosystem.

5.3.2 General Requirements

1. *Approaches for model management should be easy to use and reduce complexity; not introduce new layers of complexity.* Intuitive and straightforward approaches simplify the software engineering process, make it easier to reason about the problems being solved and are likely to be received better by the developers.

2. *Management of models should be performed using declarative languages to the extent possible.* Having declarative languages may lower the threshold for participation by non-technical stakeholders. This is because declarative languages enable a greater focus on *what* needs to be managed and *where* this management should be performed; with respect to *how* the management should be achieved.

3. *Approaches for model management should be applicable to as many (meta)-models as possible.* Useful approaches are likely to be more cost-efficient. They may also help simplifying the software engineering process because the tool set is kept smaller.

4. *Approaches for model management should be supported by development tools and editors.* The availability of stable tools is imperative for getting acceptance of an approach and for supporting real case usage scenarios.

Three overall research questions have been investigated during the work with this thesis:

RQ1: How can correct, type-safe and flexible composition and adaptation of metamodels' structure and operational semantics be achieved?

RQ2: How can evolution of metamodels and corresponding co-evolution of models be addressed to ensure correctness and flexible reuse?

RQ3: How can metamodel types be defined to support variance and reuse type-safely?

The research questions abstract over the details given in the problem statement. They are answered in Chapter 8.

Chapter 6

Contributions

This chapter describes the five artefacts that constitute the main contributions. We will introduce each of the artefacts and discuss their main properties. Additional details and in-depth explanations of each artefact are found in the respective research papers of Part II. The additional artefacts, as documented in research reports, will only be discussed in Chapter 7.

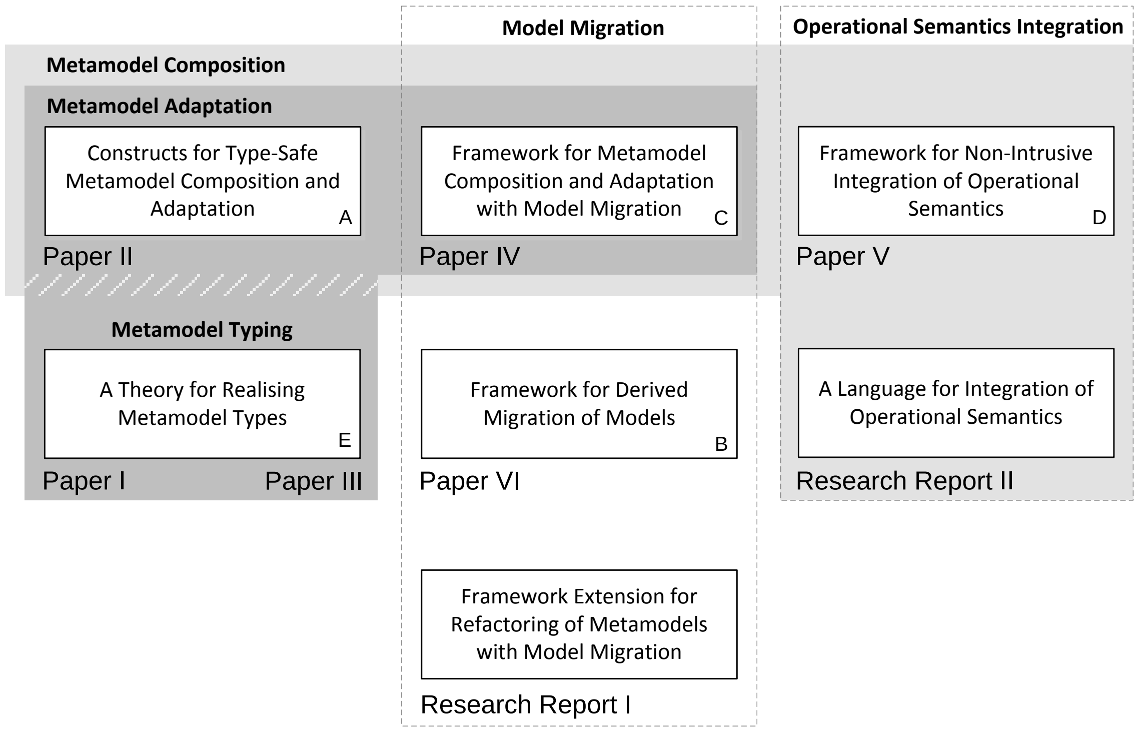

Model Migration
Operational Semantics Integration
Metamodel Composition
Metamodel Adaptation
Constructs for Type-Safe Metamodel Composition and Adaptation
A
Paper II
Framework for Metamodel Composition and Adaptation with Model Migration
C
Paper IV
Framework for Non-Intrusive Integration of Operational Semantics
D
Paper V
Metamodel Typing
A Theory for Realising Metamodel Types
E
Paper I
Paper III
Framework for Derived Migration of Models
B
Paper VI
A Language for Integration of Operational Semantics
Research Report II
Framework Extension for Refactoring of Metamodels with Model Migration
Research Report I

Figure 6.1: Overview of the artefacts and the papers in which they are discussed

Figure 6.1 gives an overview of the artefacts and in which papers they are discussed. The artefacts and papers are organised according to five topics: metamodel composition, metamodel adaptation, model migration, operational semantics integration and metamodel typing. Several of the areas overlap. For instance, Artefact C addresses metamodel composition, metamodel adaptation and model migration. It is described in detail in Paper IV.

We have created a graphical notation for illustrating how the artefacts operate. The notation is inspired from current conventions and standards, and comprises two types of concepts. The first type of concepts comprises structural units, e.g. templates, metamodels, classes, class properties and so forth. The other type of concepts includes relations, operations/transformations and mappings that can be applied on the structural units, e.g. metamodel adaptation or class merging. The capabilities of each artefact can be described in terms of a selection of relations, operations and/or mappings. The graphical notation for the structural units is given in Figure 6.2.

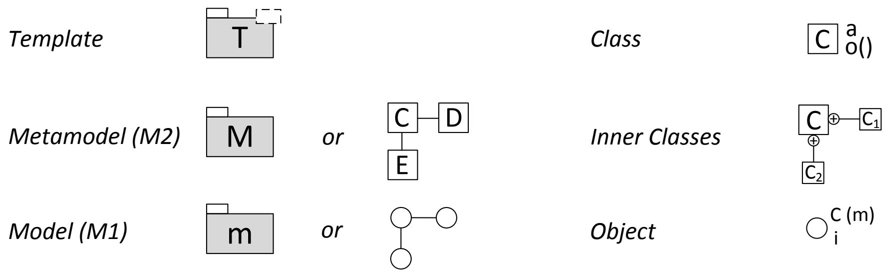


Figure 6.2: The notation for the structural units

In Figure 6.2, the C class is specified with one attribute named a and one operation named o. This indicates that the a attribute and o() operation are of special interest. It does not mean that the class does not have additional content. The same applies for inner classes, e.g. the classes C_1 and C_2 are two inner classes of C that are emphasised. An object has a type and an identifier (index). The identifier is needed to differentiate between several objects of the same class. The model in which the object reside is given in parentheses.

Figure 6.3 gives an overview of the relations and low-level operations supported by the graphical notation. The left column shows the notation for describing how a template is instantiated in another template (or package), and two types of transformations: metamodel adaptation/composition and model migration. Furthermore, the notations for model conformance, classification and subtyping are shown.

The right column of the figure gives five low-level (atomic) operations; here applied on templates. Notation wise, the operations may be applied on the other structural entities as well, e.g. directly on metamodels. High-level operations can be achieved by sequencing low-level operations.

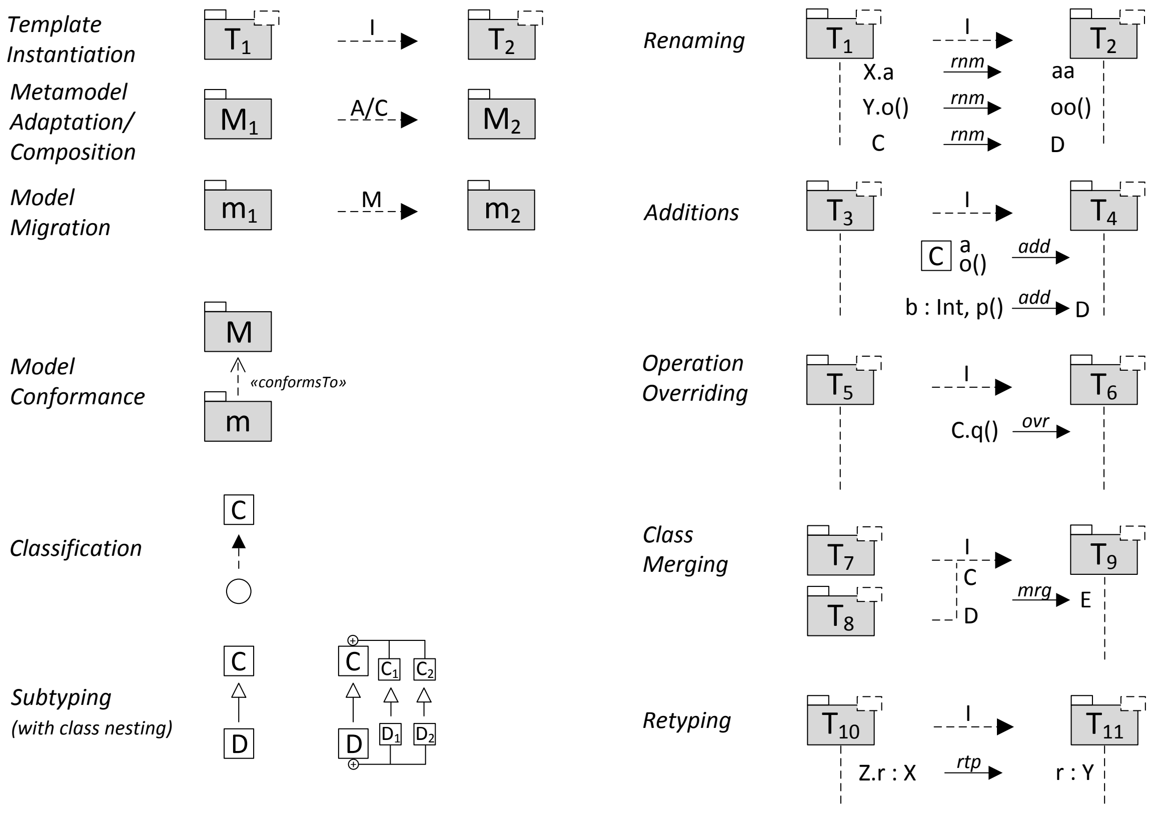


Figure 6.3: Relations and low-level operations

In the figure, three renaming operations are performed during instantiation of the T_1 metamodel template in the T_2 template. (We will discuss metamodel templates later.) First, an attribute named a in the X class of the T_1 template is renamed to aa. Second, an operation named o in the Y class is renamed to oo. Third, the class C in T_1 is renamed to D. Additions follow a similar scheme. The new contents is shown to the left of add arrows. A class C with (at least) an attribute named a and an operation named o is added to the T_4 template. An attribute named b of type Int and an operation named p is added to the D class of T_4. Types are specified when it is of relevance, e.g. as seen for the b attribute. Moreover, the operation q() found in the C class in T_5 is overridden during instantiation of this template in T_6. The class C from T_7 is merged with the D class from T_8 during instantiations of these templates in T_9. The resulting class is given the name E. Finally, the type for the r reference in the Z class of T_{10} is given the type Y instead of X.

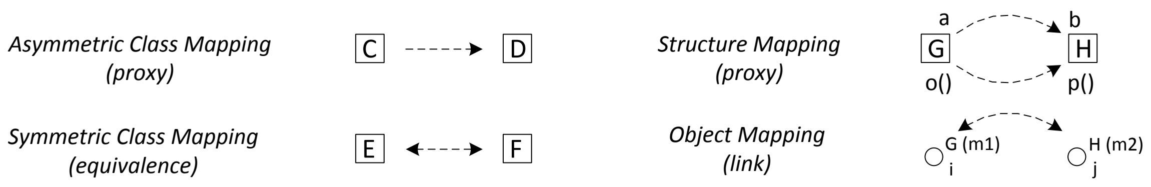

Asymmetric Class Mapping (proxy)
C
D
Symmetric Class Mapping (equivalence)
E
F
Structure Mapping (proxy)
a
b
G
H
o()
p()
Object Mapping (link)
G (m1) i
H (m2) j

Figure 6.4: Class-, structure- and object mappings, and structure export/import

Figure 6.4 gives a set of mappings. A class may be a placeholder (proxy) for another class, i.e. C is a proxy for D. In the figure, E and F are equivalent in terms of structural properties. Moreover, the a attribute of G is mapped to the b attribute of H, while the o() operation is mapped to the p() operation. That is, the attributes and operations are structurally equivalent, respectively. Finally, there is a mapping between a G object and an H object.

We will use two examples to explain the artefacts. The first example uses a metamodel (i.e. a language) for modelling of state machines, whereas the second example uses a metamodel for modelling simple arithmetic expressions. The metamodels are introduced later.

6.1 Artefact A: Constructs for Type-Safe Metamodel Composition and Adaptation

We have defined a set of new language constructs for composition and adaptation of metamodels that address limitations of current approaches for metamodel composition. Most of the constructs are closely based on the package template mechanism [95][96][97], and we therefore refer to them collectively as the metamodel template mechanism. Metamodel templates support both purely structural metamodels and metamodels that define operational semantics by means of class operations.

6.1.1 Overview

The metamodel template mechanism allows defining metamodels as reusable templates. A metamodel template contains the definition/blueprint of an EMOF-compatible class model/metamodel. Instantiation of a template causes the metamodel defined in the template to be produced and added to the context in which the instantiation occurs (local copy). The context is either another template or a package. A metamodel template may be instantiated several times in the same context. During instantiation, the classes of the metamodel may be adapted using a combination of *instantiation directives*. Merging classes from different templates is also supported. An instantiation directive is specified in a textual concrete syntax reflecting a specific construct/operation. A template instantiation (i.e. each directive, except for retyping) preserves type-safety of the enclosed metamodel. Each instantiation of a template gives a unique metamodel with its own distinct set of regular Kermeta classes, which are not related to classes produced in another instantiation. When templates are instantiated in other templates we get a tree-like structure we refer to as a template hierarchy.

Metamodel composition is achieved by instantiating two or more templates in the same context and then merging two or more classes from the different metamodels. Alternatively, the metamodels may be composed by establishing references between classes of the metamodels. This include creating new classes (interfacing) that represent conceptual bridges between two metamodels. It is also possible to use subtyping between classes of different metamodels. There are two types of adaptation possible. The first type of adaptation is concerned with the structure of a metamodel. Specifically, it is possible to rename classes, properties (i.e. attributes and references) and operations. Moreover, new classes may be added to a metamodel and new properties and operations may be added to the existing classes. It is also possible to change the type of a reference. The second type of adaptation relates to the operational semantics of the metamodels (as defined in class operations). This includes the ability to override operations and adding code by creating new operations. This in turn supports composition and adaptation of operational semantics.

Initially we used the same syntax for the instantiation directives as used by the package template mechanism. However, it may be argued that this syntax is difficult to read. We therefore invented a more straightforward syntax for some of the directives that we will use to explain the mechanism[1]. The constructs/instantiation directives may be divided into two groups. The first group comprises constructs derived directly from the package template mechanism. The second group consists of constructs designed specifically for application on metamodels. Tables 6.1 and 6.2 give an overview of the two groups of constructs.

Construct	Original Syntax	Revised Syntax	Description
A1	`inst T [with]`	`instantiate(T, ...)`	Template instantiation
A2	`C1 => C2`	`C1 -> C2`	Class renaming
A3	`p1 -> p2`	`p1 -> p2`	Property renaming
A4	`o1() -> o2`	`o1() -> o2`	Operation renaming
A5	*implicit*	`merge(C1, C2, ..., Cn, D)`	Class merging
A6	`C adds {...}`	`codeblock cb {...} add(C, cb)`	Addition of code

Table 6.1: Constructs modelled on package template operations

Kermeta is a language and framework/workbench for engineering of metamodels [12]. It allows specifying both the structure and semantics of metamodels. Kermeta does not contain language constructs for composing metamodels (except for its open classes nature), and we found it to be a good candidate for illustrating the new constructs for composition and adaptation of metamodels.

[1]The implemented prototype uses the original version of the syntax.

Construct	Original Syntax	Revised Syntax	Description
B1	`inst t1:...`	`instantiate(..., t1)`	Namespace declaration
B2	`r1 :-> t1::C`	`retype(r1, t1.C)`	Retyping
B3	`group C1, C2, ..., Cn -> P`	`group(C1, C2, ..., Cn, P)`	Class grouping

Table 6.2: Additional constructs

In the following, we treat the constructs as an extension of the Kermeta language, and use Kermeta to illustrate the mechanism. This corresponds to a prototype tool we have created that works on a subset of Kermeta.

The discussion of the mechanism is organised in six topics: *basic metamodel extension and adaptation*, *propagation of names*, *symmetric composition and composition of semantics*, *full static type-checking and preservation of type-safety*, *template hierarchies* and *traceability*.

6.1.2 Basic Metamodel Extension and Adaptation

We will illustrate how a template with a metamodel for modelling of state machines can be extended and adapted for modelling of state machines with weighted transitions (including adaptation of the operational semantics). A weighted state machine allows adding a weight between 0 and 1 (including endpoints) to each transition of a state machine [100]. The weight associated with a transition indicates the probability of this transition being triggered. The sum of the weights of all the outgoing transitions from a state must add up to 1. We consider a variant of the weighted state machine that allows the sum of all the weights to be less than 1 (but never more than 1). This means that a transition is not always triggered when an event is received (and the current state is not changed). The semantics of the added language functionality is as follows. Every transition is associated with an event. A given weighted transition is triggered if two conditions are fulfilled. First, the state machine must receive an event that is associated with one of the transitions of the current state. Second, the weight/probability of the transition has to be greater than or equal to a random generated floating point number in the interval [0,1]. A regular transition triggers if the first condition is fulfilled.

Figure 6.5 gives an overview of the case by using the graphical notation previously introduced. The figure shows how the templates TStateMachine and TWeightedTransition are instantiated and combined to create the metamodel named M in the stateMachine package. Four directives are used to achieve this: the step(...) operation of the State class is overridden with a new definition (*ovr*), the Transition and WTransition classes are merged and given the new name WeightedTransition (*mrg*), the val attribute of the Weight class is renamed to probability (*rnm*), and an attribute named description and an operation named triggerWeighted are added to the WeightedTransition class (*add* twice). Three consecutive dotes (...) represent additional code (or structural elements) that has been omitted from the figure. New and changed elements are shaded for easier reference. (The names of most references are excluded for clarity.)

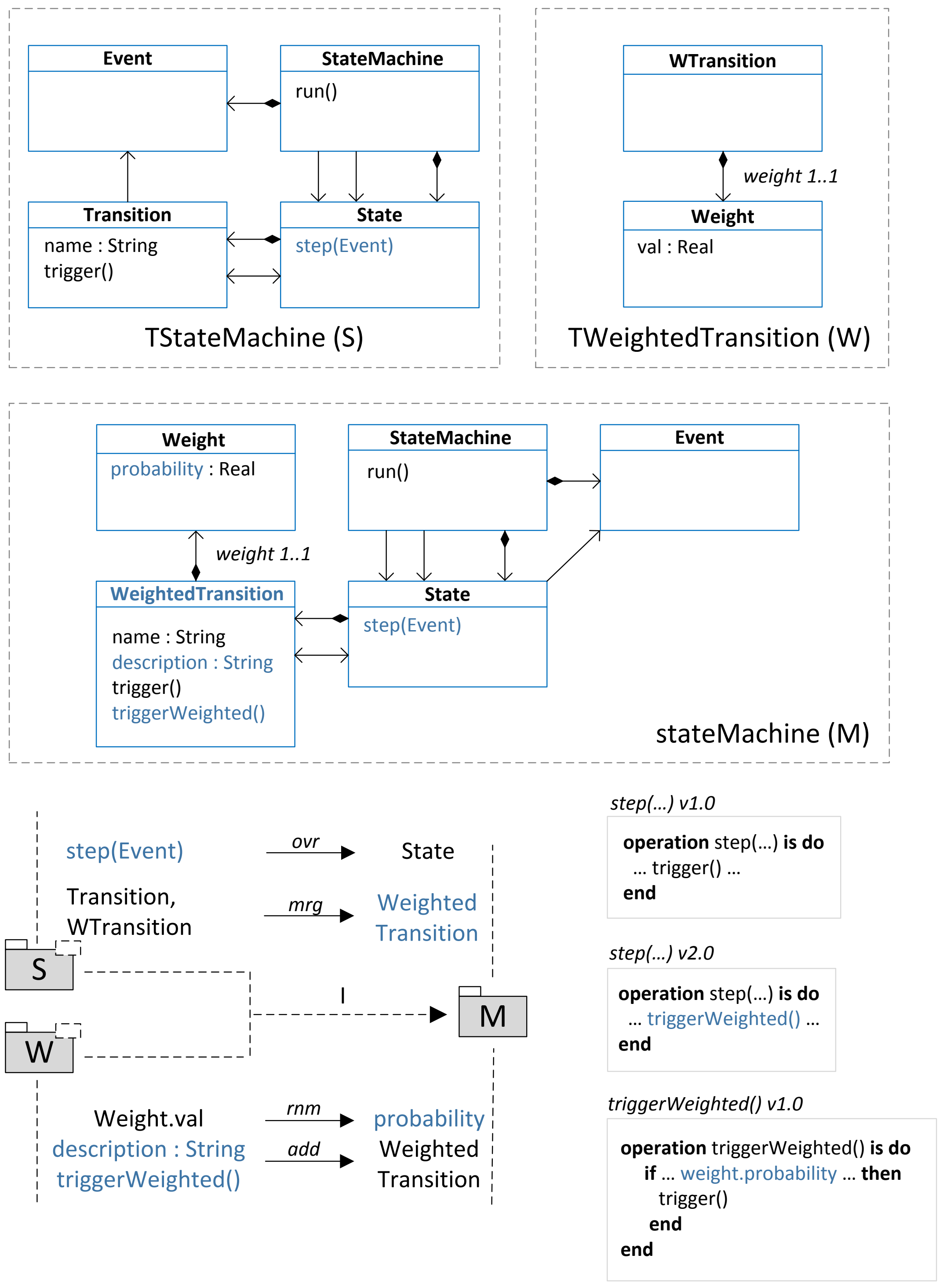


Figure 6.5: Composing and adapting metamodels

```
// TStateMachine.kpt
template TStateMachine
{
  class StateMachine {
    attribute states : State[1..*]#stateMachine
    attribute events : Event[1..*]
    reference currentState : State[1..1]
    reference initialState : State[1..1]
    operation run() is do ... end
  }
  class State {
    reference stateMachine : StateMachine[1..1]#states
    attribute outgoing : Transition[0..*]#source
    reference incoming : Transition[0..*]#target
    operation step( event : Event ) is do
      var target : Transition
      target := outgoing.select{ t | event.equals( t.event ) }
      if target != void then
        target.trigger()
      end
    end
  }
  class Transition {
    attribute name : String
    reference event : Event[1..1]
    reference target : State[1..1]#incoming
    reference source : State[1..1]#outgoing
    operation trigger() is do
      source.stateMachine.currentState := target
    end
  }
  class Event { ... }
}
```

Figure 6.6: Template for modelling of state machines

Figure 6.6 gives an excerpt of the template containing the metamodel for modelling of state machines, whereas Figure 6.7 gives the other template containing classes for modelling of weighted transitions, i.e. the classes WTransition and Weight. Notice that Kermeta uses the keyword reference for references/associations and attribute for class attributes (of primitive types) and containment references. We use the revised syntax for the instantiation directives in the example.

```
// TWeightedTransition.kpt
template TWeightedTransition
{
  class WTransition
  {
    attribute weight : Weight[1..1]
  }

  class Weight
  {
    attribute val : Real
  }
}
```

Figure 6.7: Template containing classes for modelling of weighted transitions

```
package stateMachine;
require "TStateMachine.kpt"
require "TWeightedTransition.kpt"

instantiate( TStateMachine, sm );
instantiate( TWeightedTransition, wt );
wt.Weight.val -> probability;
merge( sm.Transition, wt.WTransition, WeightedTransition );

codeblock c1
{
  // Overrides the existing step() operation when added to State
  operation step( event : Event ) is do
    var target : Transition
    target := outgoing.select{ t | event.equals( t.event ) }
      if target != void then
        target.triggerWeighted()
      end
  end
}

codeblock c2
{
  attribute description : String

  operation triggerWeighted() is do
    // generates a random number between 0 and 1
    var randomNumber : Real init ...
    if weight.probability >= randomNumber then
      trigger()
    end
  end
}

add( State, c1 );
add( WeightedTransition, c2 );
```

Figure 6.8: Template instantiation

Figure 6.8 shows instantiation of the two templates within a package named stateMachine. instantiate(TStateMachine, sm) initiates instantiation of the TStateMachine template and declares sm as a namespace identifier. The namespace identifier is later used to reference the classes of the template for the specific instantiation (several instantiations of the same template yield several namespace identifiers). Similarly, wt is declared as a namespace identifier for the content of the TWeightedTransition template. wt.Weight.val -> probability renames the val attribute of the Weight class to probability. The Transition and WTransition classes from the two templates are merged and given the new name WeightedTransition using the merge directive.

```
package stateMachine;

class StateMachine {
  attribute states : State[1..*]#stateMachine
  attribute events : Event[1..*]
  reference currentState : State[1..1]
  reference initialState : State[1..1]
  operation run() is do ... end
}
class State {
  reference stateMachine : StateMachine[1..1]#states
  attribute outgoing : WeightedTransition[0..*]#source
  reference incoming : WeightedTransition[0..*]#target
  operation step( event : Event ) is do ... triggerWeighted() ... end
}
class WeightedTransition {
  attribute name : String
  attribute description : String
  reference event : Event[1..1]
  reference target : State[1..1]#incoming
  reference source : State[1..1]#outgoing
  attribute weight : Weight[1..1]

  operation trigger() is do ... end

  operation triggerWeighted() is do
    // generates a random number between 0 and 1
    var randomNumber : Real init ...
      if weight.probability >= randomNumber then
        trigger()
    end
  end
}
class Weight {
  attribute probability : Real
}
class Event { ... }
```

Figure 6.9: Metamodel for modelling of weighted state machines

Two code blocks are then defined. codeblock c1 contains a new definition for the step(...) operation in the State class. The step(...) operation is overridden when this code block is later added to the State class. That is, only the overridden operation is included in the State class after instantiation. The new definition of the step(...) operation invokes the triggerWeighted() operation which is defined in codeblock c2 and later added to the WeightedTransition class. triggerWeighted() evaluates the value of the probability attribute of the Weight class to determine whether a transition should be triggered. An attribute named description of type String is also specified in codeblock c2. The package containing the resulting metamodel is given in Figure 6.9. Notice how the outgoing attribute and the incoming reference in the State class are now typed with the WeightedTransition class.

6.1.3 Propagation of Names

In the example, the val attribute of the Weight class was given the new name probability. We also gave the name WeightedTransition to the class resulting from merging the Transition and WTransition classes. Giving a new name to a merged class can be seen as a renaming as well. Renaming is not constrained to the contents of a single template. Instead, renaming an element affects the contents of all templates of a hierarchy that reference or use the element. That is, the new name replaces all names the element previously had resulting from earlier applications of the rename and merge directives. Renaming is not resolved before instantiation which means that the name for a given element is decided first at the very last moment. An example will illustrate this.

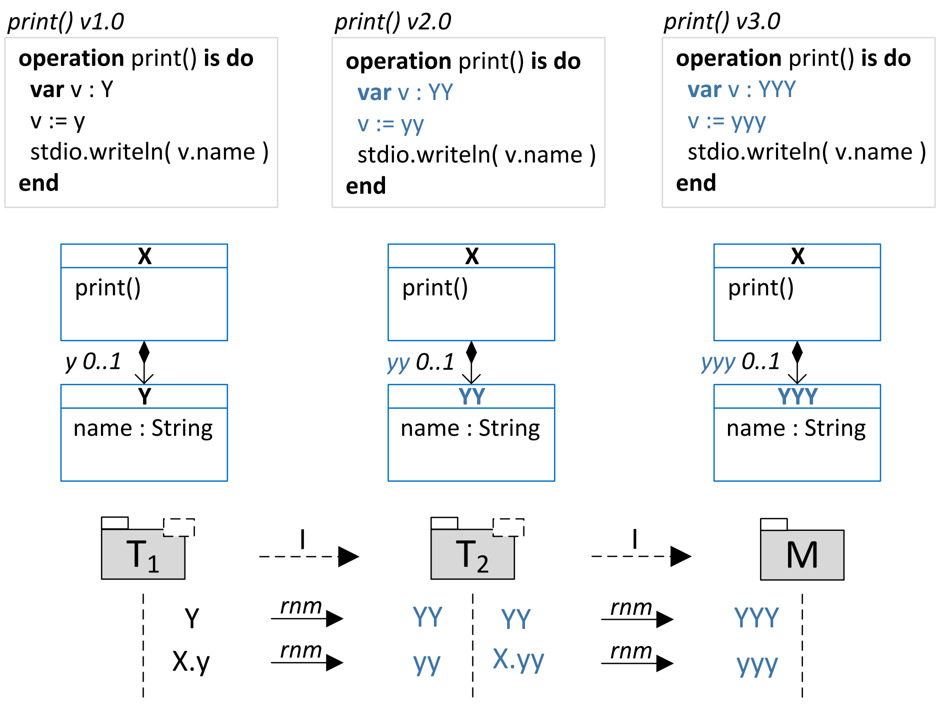


Figure 6.10: Propagation of names during instantiation

Figure 6.10 gives a template hierarchy comprising two templates and a package/metamodel: T1, T2 and M. T2 contains an adapted version of the metamodel defined in T1, i.e. T1 is instantiated by/inside T2. T2 is instantiated in a package giving the metamodel M. Four renaming operations are performed during the instantiations. The shaded code illustrates how the content of the print() operation changes to reflect the renaming operations. That is, the code shows the definition of the print() operation at different locations in the hierarchy. As an example, v2.0 of print() is produced if the T2 template is instantiated in a package without further renaming operations. To be clear, the print() operation is only defined once, yet its code reflects the renaming directives used in the instantiations.

```
template T1 {
  class X {
    attribute y : Y[0..1]

    operation print() is do
      var v : Y
      v := y
      stdio.writeln( v.name )
    end
  }
  class Y {
    attribute name : String
  }
}

template T2 {
  instantiate( T1, t1 );
  t1.Y -> YY;
  t1.X.y -> yy;
}

package p {
  instantiate( T2, t2 );
  t2.YY -> YYY;
  t2.X.yy -> yyy;
}
```

Figure 6.11: Implementation of the templates and the instantiations

```
package p;

class X {
  attribute yyy : YYY[0..1]

  operation print() is do
    var v : YYY
    v := yyy
    stdio.writeln( v.name )
  end
}
class YYY {
  attribute name : String
}
```

Figure 6.12: The resulting package after the final instantiation

Figure 6.11 gives the definitions of the templates and the package. The T2 template instantiates the T1 template and gives the Y class the new name YY, whereas the y attribute of the X class is renamed to yy. The p package instantiates the T2 template and renames the same class and attribute again: the YY class is now renamed to YYY and the yy attribute of the X class is given the name yyy. Figure 6.12 shows the resulting p package after the final instantiation.

As can be seen in Figure 6.12, the content of the p package is structurally the same as the content of the T1 template. However, a new name is given to one of the classes and one of the attributes. The new names are propagated to all places where the old names were used, e.g. in the print() operation. In general, a new name is also propagated to statements within operations, e.g. if and loop statements, expressions like class instantiation (new) and variable and operation declarations. The name propagation occurs at an arbitrary number of nested scopes. Propagation of names works in the same way when classes are merged and thereby given a new name. Consequently, all code is updated when elements are renamed or classes are merged.

When the metamodels define operational semantics the value of name propagation becomes even more evident. Consider the print operation of the X class in Figure 6.11. As we have seen, renaming resulted in a new name and type (i.e. the type name is new) for the y attribute. If these renamings are not reflected in the statements of the operation, the code of the operation will no longer be syntactical correct. As can be seen, renaming preserves type-safety. It ensures that the structure of metamodels is well-formed and that the combination and adaptation of operational semantics (when the metamodels specify such) do not result in type errors.

```
template T1
{
  class X {
    operation o() is do
      var y : Y init Y.new
    end
  }
  class Y {}
}

template T2
{
  class Z {}
}

package p {
  instantiate( T1, t1 );
  instantiate( T2, t2 );
  merge( t1.Y, t2.Z, YZ );
}
```

Figure 6.13: Merging of two classes with propagation of name

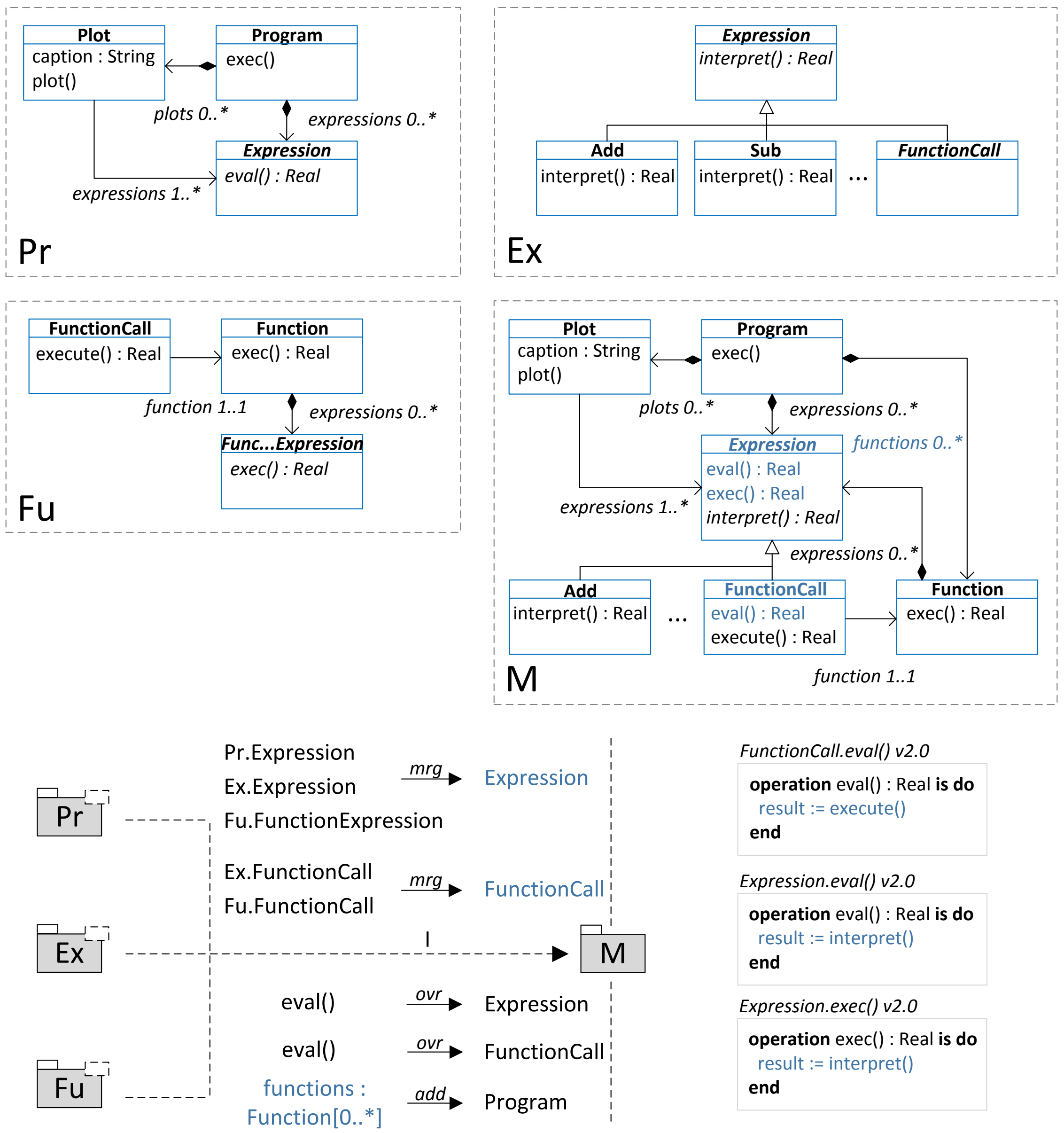
Figure 6.13 shows how the new name for a merged class propagates to a variable declaration statement and a class instantiation expression. The result of the instantiations is given in Figure 6.14.

```
package p;

class X
{
  operation o() is do
    var y : YZ init YZ.new
  end
}

class YZ {}
```

Figure 6.14: The resulting package

6.1.4 Symmetric Composition and Composition of Semantics

The metamodel template mechanism supports symmetric composition of an arbitrary number of metamodels, e.g. by merging classes or by creating references between classes of different metamodels. Symmetric composition means that all concerns are equal with respect to how they can be combined [92].



```
operation eval() : Real is do
  result := execute()
end
```

```
operation eval() : Real is do
  result := interpret()
end
```

```
operation exec() : Real is do
  result := interpret()
end
```

Figure 6.15: Creating a metamodel for plotting of mathematical expressions

Composition mechanisms that only support asymmetric composition are not ideal in situations where several metamodels have constructs whose semantics overlap. First, these mechanisms typically need to be used repeatedly in order to compose the metamodels. Second, composing the operational semantics (if the metamodels provide such by class operations) may be more difficult. Consider the three templates of Figure 6.15. The Pr template comprises a metamodel for modelling of simple plots of mathematical expressions. Ex contains a metamodel for modelling of simple mathematical expressions, whereas Fu contains a metamodel for modelling of parameterless functions. The metamodels of the templates are combined to create a new metamodel for creating and plotting mathematical expressions with functions.

```
template Pr {
  class Program {
    attribute expressions : Expression[0..*]
    attribute plots : Plot[0..*]

    operation exec() is do ... end
  }
  abstract class Expression {
    operation eval() : Real is abstract
  }
  class Plot {
    reference expressions : Expression[1..*]
    attribute caption : String
    operation plot() is do ... end
  }
}

template Ex {
  abstract class Expression {
    operation interpret() : Real is abstract
  }

  class Add inherits Expression { ... }
  class Sub inherits Expression { ... }
  class Multi inherits Expression { ... }
  class Number inherits Expression { ... }
  abstract class FunctionCall inherits Expression { ... }
}

template Fu {
  abstract class FunctionExpression {
    operation exec() : Real is abstract
  }
  class Function {
    attribute expressions : FunctionExpression[0..*]
    operation exec() : Real is do ... end
  }
  class FunctionCall {
    reference function : Function[1..1]
    operation execute() : Real is do ... end
  }
}
```

Figure 6.16: Source templates

The definitions of the templates are given in Figure 6.16. As can be seen, all the metamodels have a class for describing an expression. A natural choice is therefore to compose the three metamodels by merging the expression classes. In addition, the two classes for expressing function calls are logical candidates for merging. Since we allow symmetric composition all the metamodels can be composed simultaneously.

```
package m {
  instantiate( Pr, pr );
  instantiate( Ex, ex );
  instantiate( Fu, fu );

  merge( pr.Expression, ex.Expression, fu.FunctionExpression,
    Expression );
  merge( ex.FunctionCall, fu.FunctionCall, FunctionCall );

  codeblock c1 {
    operation eval() : Real is do
      result := interpret()
    end

    operation exec() : Real is do
      result := interpret()
    end
  }

  codeblock c2 {
    // Overrides the inherited eval() operation from Expression
    // when added to FunctionCall
    operation eval() : Real is do
      result := execute()
    end
  }

  codeblock c3 {
    attribute functions : Function[0..*]
  }

  add( Expression, c1 );
  add( FunctionCall, c2 );
  add( Program, c3 );
}
```

Figure 6.17: Constructing the composite metamodel

Figure 6.17 shows how composition of the metamodels is achieved using the metamodel template mechanism. Figure 6.18 gives the resulting package m after instantiating the templates. Merging an abstract class with a concrete class gives a concrete merged class. A merged class is only abstract if all the constituent classes, as referred to in the merge operation, are abstract. Overriding an abstract operation always gives a concrete operation as result.

As a comparison, let us consider how things play out if we use an asymmetric composition mechanism, i.e. that can only compose two metamodels simultaneously. A possible approach is to first treat the metamodel of T1 as the base metamodel and compose this with the metamodel of the T2 template which takes an aspect role. The resulting metamodel can then be regarded as a base metamodel and be composed with the metamodel of T3. We could also start by first composing the metamodels of T2 and T3, and then compose the resulting metamodel with the metamodel of T1. Regardless of how the metamodels are composed, we are required to use the composition mechanism two times in order to compose the metamodels. Additional metamodels (other than the three listed) would have complicated the composition further. On the other hand, symmetric composition with support for composing an arbitrary number of metamodels simultaneously increases agility by handling the composition of all metamodels in one single step.

```
package m;

class Program {
  attribute functions : Function[0..*]
  ...
}
abstract class Expression {
  operation eval() : Real is do
    result := interpret()
  end

  operation exec() : Real is do
    result := interpret()
  end

  operation interpret() : Real is abstract
}
class Plot { ... }
class Add inherits Expression {}
...
class Function {
  attribute expressions : Expression[0..*]
  operation exec() : Real is do ... end
}
class FunctionCall inherits Expression {
  reference function : Function[1..1]
  operation execute() : Real is do ... end
  operation eval() : Real is do
    result := execute()
  end
}
```

Figure 6.18: The resulting metamodel after composition and adaptation

```
package p;
...

codeblock c1 {
  operation eval() : Real is do
    if self.isKindOf( FunctionCall ) then
      result := execute()
    else
      result := interpret()
    end
  end
}

add( Expression, c1 );
```

Figure 6.19: Composition of operational semantics

Composing the operational semantics of metamodels may require symmetric composition. As an example, let us assume that we would like to override the eval operation of the Expression class only once, and not twice as in Figure 6.17. This requires that all metamodels are composed simultaneously. Figure 6.19 illustrates this.

6.1.5 Full Static Type-Checking and Preservation of Type-Safety

An important feature of metamodel templates is the ability to preserve the type-safety of metamodels when they are adapted and composed, i.e. that the application of the instantiation directives is type-safe. When metamodels contain operational semantics, it is essential that this can be guaranteed statically, i.e. when the templates are instantiated. All template directives, except for retyping, preserve type-safety. This means that the operational semantics of a metamodel will be type-safe also after the metamodel has been composed, and/or adapted, with another metamodel, or metamodels, as long as the operational semantics of the two metamodels were type-safe in the first place.

In [98], the authors show that package template programs (utilising a core version of package templates) are consistent. A package template program is a combination of templates and packages (resembling what we refer to as a template hierarchy). Templates and packages are either *closed* or *open*. Closed templates and packages do not instantiate other templates, while open templates and packages do. The content of a closed template or package is a subset of Java. A closed template (or package) is therefore semantically complete and self-contained (as long as contents of external packages are not referenced), which means that its content can be type-checked by a Java compiler.

The authors define four transformations (*fortifying*, *renaming*, *addition handling* and *composing*) that implement the semantics of the package template operations (directives). The transformations are semantics-preserving, i.e. they preserve the intention and meaning of a package template program, with the exception of renamed elements, shadowed variables and overridden methods. The authors show that a template program comprising an open package and a set of open and closed type-safe templates can be transformed to a closed type-safe package, which means that the package template operations are consistent and preserve type-safety.

The same reasoning can be applied to the metamodel template mechanism. A closed template comprises a metamodel that can be type-checked by the underlying metamodelling language, e.g. Kermeta. The directives preserve type-safety if an open package or template, and a set of open and closed type-safe templates, can be reduced to a closed template or package whose metamodel is type-safe according to the metamodelling language. An important observation is that renaming is done sequentially. This implies that an element can not be given the same name of an existing element (of the same type). An existing name can first be used again on another element if the element that originally had the name has been given a new name. By construction, this also means that all references to the old name have been updated. Using the old name on a new element would now be safe.

The work of [98] shows that this is the case for the directives that are based on the package template operations. However, metamodel templates differ in two major ways from package templates: the metamodel mechanism provides additional and alternative instantiation directives and the mechanism operates on EMOF/Ecore models (metamodels) and not Java programs. This has to be taken into consideration.

Metamodel Templates Versus Package Templates Metamodel templates use namespace identifiers for accessing the template classes. This increases flexibility during instantiations and allows using a dedicated merge directive instead of achieving class merging implicitly by giving two or more classes the same name, as is the case with package templates. It also allows specifying code blocks whose content can be added to several classes. The metamodel template mechanism also provides a retyping directive and a directive for grouping of classes.

Using namespace identifiers does not conflict with the pre-conditions and rules specified for the transformations in [98]. The reason for this is that using namespace identifiers only decouples directives with respect to how the directives are specified using the package template notation. The directives can easily be rewritten to the more condensed form as used by the package template mechanism[2]. This includes rewriting of the dedicated directive for class merging.

Retyping works by giving a new type to a reference. The retyping directive was designed to give users a light-weight approach for establishing a reference between classes of different metamodels. The new reference type has to be a subtype of the previous reference type by construction; all class features and operations (of the old reference type) are therefore still accessible after retyping. However, the directive is not type-safe with regard to write operations. An example of an unsafe operation that causes a type error is when an instance of the old reference type is added to the reference (which now has been retyped to a subtype). The directive for grouping of classes creates a new package in which the specified classes are added. Since classes can be referenced across package boundaries, this directive is type-safe. Finally, a code block, used in combination with an addition directive, is semantically equivalent to an addition class in the package template mechanism. The content of a code block may be added to several classes by using more addition directives. However, this is the same as specifying several addition classes (all with the same content) using the package template notation. Consequently, code blocks are type-safe according to [98].

Another difference with metamodel templates, with respect to package templates, is that metamodel templates are not instantiated at compile-time. A package template is instantiated during compilation which means that the classes from the template instantiation are added directly in the generated Java byte code. Metamodel

[2]This form is also used by the implemented metamodel template prototype.

templates operate on a higher abstraction level. This means that template instantiation is resolved as an independent step. That is, the template instantiation produces a specific metamodel, i.e. a set of regular Kermeta classes, which can be used for modelling. Compilation or interpretation of the Kermeta code is first performed before/when a model executes.

Metamodels Versus Programs Metamodels relate classes using bi-directional (containment) references/associations which the package template mechanism is not designed for. A reference also has a multiplicity which specifies the range (number) of objects it may reference or contain.

Bi-directional references do not introduce type errors as long as the renaming of a reference is reflected in the opposite reference (also known as *opposite property* in MOF[3]) of the opposite class (if such is specified). The opposite reference does not have to be unique since this is not a property of the opposite class. In other words, several references in a class may have equally named opposite references. This means that it is not possible to induce naming conflicts for opposite references. Using containment references and different multiplicities do not cause any problems since these properties/constraints are not altered by the directives.

6.1.6 Template Hierarchies

Since templates may be instantiated within each other it is possible to create template hierarchies. A template hierarchy is a graph consisting of nodes in the form of templates, and optionally a package, and template instantiations comprising the edges of the graph. Paper VI describes this structure more carefully. The main achievement with template hierarchies is that metamodel evolution can be addressed in steps, where a template may be defined as a direct derivative of other templates through instantiations; and consequently forming branches. Metamodel elements acquired through a branch, i.e. elements defined in the templates of a branch, can be adapted regardless of what templates these elements are defined in. Moreover, metamodels can be built by using already defined metamodel templates. Each template of a template hierarchy can be instantiated which produces a specific metamodel variant or version. That is, all previous variants and versions of a metamodel are available. A template hierarchy therefore naturally maintains traceability information on how metamodels evolve.

6.1.7 Traceability

Available metamodel composition mechanisms produce a composed metamodel from two or more metamodels. However, these mechanisms do not have much support for internal traceability, i.e. traceability between different versions or variants of

[3]http://www.omg.org/mof

models [92]. One reason for this is that the source metamodels can typically not be recovered easily from the composed metamodel. This requires backtracking changes by inspecting source metamodels and related files (e.g. a weaving model). Metamodels defined in templates evolve in hierarchies. Thus, changes can easily be traced by navigating a branch in a template hierarchy. Specifically, the instantiation directives traversed by navigating the branch describe all changes applied to a given metamodel and can collectively be seen as a difference model, i.e. a model that describes the primitive changes between two metamodel versions [71].

6.1.8 Design Decisions and Limitations

The metamodel template mechanism supports composition and adaptation of metamodels by applying a sequence of pre-defined metamodel-specific instantiation directives. The metamodel-specific nature of the directives is caused by how the application of a directive requires specifying what specific metamodel element(s) that the directive should work on. Because of this, a template instantiation (including the directives) can not be reused or applied for other templates and therefore not reflect evolutionary patterns that may apply to several metamodels. The directives are essentially parallel independent [70]; conflicts occurring during class merging need to be addressed using renaming directives prior to the merge directive. In such case, the merge directive is dependent on one or more renaming directives.

A limitation of the mechanism is the limited number of available directives. In particular, there are no destructive directives [58], e.g. it is not possible to specify that an element should be deleted. There are three reasons backing the decision of having a reduced set of directives.

First, one of the major goals and achievements of metamodel templates is the ability to support type-safe composition and adaptation. Directives for e.g. deleting elements complicate how preservation of type-safety can be achieved. The code of an operation may refer to any attribute, reference or operation of the class in which the operation is defined. Moreover, it is possible for an operation to refer any attribute, reference or operation of all classes that can be navigated by references. The running state machine example illustrates this (see Figure 6.6). This means that we would have to make sure that all references to a deleted element (attribute, reference or class) or a moved element (attribute, reference or operation)[4] in the code are identified and notified to the user. This may be achieved by a thorough analysis of the code of every operation that may potentially refer to a deleted or moved element, i.e. the latter also requires that the code of the class, to which an attribute or reference is moved, is updated to utilise the new (moved) attribute or reference. The result of the analysis would be a number of markers that identify places in the code that need to be revised by the user. A disadvantage with manually updating the code is the risk of changing the metamodel semantics and introduce errors. In

[4]Moving elements in a class hierarchy may in some cases preserve type-safety.

the metamodel template mechanism we reduce this risk by only allowing adding new/updating the code by overriding operations, i.e. all changes are contained within overridden operations. This localisation of the changes makes it easier to identify potential anomalies and errors that may occur during execution of the code following adaptation. However, overriding operations can not address necessary code changes well when elements are deleted or moved. (Complete new code may be specified by overriding an operation, but it is not a practical solution.) It is possible to disregard the goal of preserving type-safety. However, this will counteract one of the major achievements using metamodel templates.

Merging of attributes and references is another directive that is not supported because of the goal of preservation of type-safety; since code may (are likely to) refer to the attributes and references. In addition, attributes of different classes that appear identical may indeed have different semantics [43], which makes merging of these attributes questionable. Following a similar argumentation, merging of operations is not supported. Also, operations (may) contain code which makes merging of operations an open question, i.e. it is not evident how the code of the different operations can be combined.

Second, the mechanism supports the majority of the most common metamodel adaptations, which according to different sources are: addition of properties [53], modifications of elements, e.g. renaming of classes and attributes [60] and addition, deletion and modification of attributes and references, e.g. modification by changing the multiplicity of a property or moving an attribute to another class [65]. Hence, even with the limited number of directives, most common adaptations can be performed. Changing the multiplicity of a property may as well compromise type-safety. The code of an operation may expect a certain number of values or objects. Changing either the lower or upper bound of a multiplicity may render such code invalid.

Third, having destructive directives makes automatic model migration more difficult. This is discussed in the next section regarding Artefact B.

Metamodels defined in templates can be composed by merging of classes or by establishing relationships (references or subtyping) between classes. All these forms of composition are symmetric. Symmetric composition ensures a high degree of agility and flexibility in how metamodels can be composed. However, being able to also compose metamodels asymmetrically is advantageous in situations when an aspect (advice) metamodel has to be composed with a base metamodel at several places. It is not obvious when to apply a symmetric or asymmetric composition approach [92], which means that having support for both kinds of approaches increases the usability of a mechanism. In MDE, asymmetric composition is typically concerned with open class and pointcut-advice mechanisms [92]. Of the two choices, using a pointcut-advice mechanism is more expressive when it comes to specifying where an aspect should be composed (woven) with the base model.

The metamodel template mechanism provides a foundation for building an asymmetric composition mechanism, or more precisely, an extension for supporting asymmetric composition. The extension may be created as a pre-processor that accepts a mixture of templates and specifications of pointcuts. A pointcut can be defined using an expression language or by modelling the pointcut as a metamodel fragment representing a specific structure and names that must be matched in the base metamodel. These ways of defining pointcuts are used by XWeave [31]. Using pointcuts allows applying the same aspect metamodel to all places in a base metamodel that fulfill the requirements of the pointcut. The output of the pre-processor would be a set of instantiation directives, which means that the actual composition is still performed symmetrically and type-safe.

A more thorough explanation and examples using the metamodel template mechanism are found in Papers II and VI. This includes an explanation of the retyping directive.

6.2 Artefact B: A Framework for Derived Migration of Models

Adapting and/or composing metamodels using the metamodel template mechanism results in the creation of a new metamodel or metamodel variant. The mechanism works exclusively on metamodels and does not consider existing models of the source metamodels. This means that existing models have to be composed using a third-party model composition mechanism or otherwise updated in order to re-establish conformance with the new metamodel or metamodel variant. We have created a prototype framework that builds on Artefact A. The framework supports semi-automatic migration and composition of existing models. The framework generates name mappings and specifications for generating default objects and values from the template instantiation directives specified by the user. (The retyping directive does not allow semi-automatic migration and is therefore not considered.) Generating default objects and values is required for re-establishing model conformance [73][64]. The name mappings and specifications comprise basic transformations in a raw format (tuples). The basic transformations can be used as input for generating model-to-model transformations, e.g. in ATL.

6.2.1 Overview

The framework is a continuation of the metamodel template mechanism and works by utilising the user-specified template instantiation directives, with the exception of the retyping directive which is not supported. The directives are used for deriving transformations that when executed migrate or compose existing models. A *migration route* denotes a path in a template hierarchy comprising an arbitrary number of templates and two packages. The packages contain the source and target metamodels for a (potentially valid) migration transformation. For a valid migration route, models of the source metamodel may be migrated to models of the target metamodel. From the user's perspective, the process of migrating models comprises two steps. First, the user specifies the source and target metamodels of a migration by pointing out two packages in the hierarchy. The framework checks the validity of the migration route. If the migration route is valid, a number of transformations is generated. The transformations correspond to the cumulative result of all actions caused by all the instantiation directives that previously generated the two metamodels. Second, the user selects the models of the source metamodel that are intended to be migrated, which are then transformed to conformant models of the target metamodel. The process of composing models is similar with the process of migrating models. The difference is that several source metamodels are specified, which in turn requires selecting a model (and objects) of each of the source metamodels for the composition to be performed.

6.2.2 Model Migration and Composition

A template hierarchy may have several different forms, and the metamodels defined in the templates may be adapted and composed in many ways. Hence, what models that can be successfully migrated and composed are conditioned on the structure of the hierarchy and the type of instantiation directives that have been used throughout the hierarchy. A hierarchy may be decomposed in terms of three basic structures: a branch, two branches connected by a common template and merged branches.

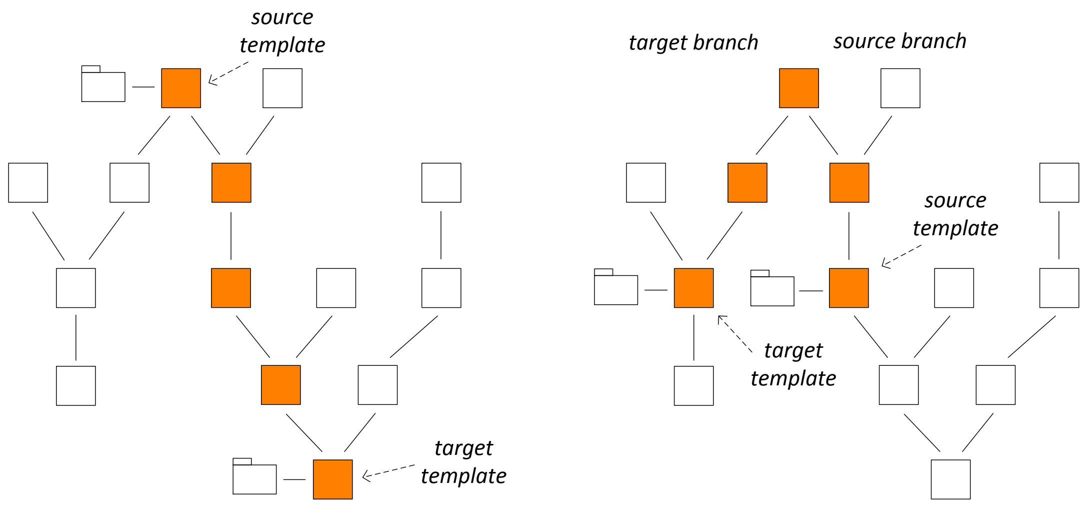


Figure 6.20: Migration routes in a template hierarchy

Model Migration

For migration of models there are only two types of migration routes that are pragmatic: a branch and two branches connected by a common template. A branch reflects how a metamodel changes linearly in variations and versions. This is the typical scenario for how a metamodel evolves. A metamodel may also evolve in two or more parallel versions which is reflected by the other type of migration route. However, for model migration to make sense for this kind of evolution, only renaming of elements (and overriding of operations) can be used in the instantiations.

Figure 6.20 shows examples of two migration routes, one of each kind, in a template hierarchy (the notation for the templates is simplified). The source and target templates are instantiated in packages giving a source and target metamodel, respectively.

Model Composition

Model composition only makes sense when two or more templates are instantiated in a template or package, yielding a number of merged branches. Transformations for model composition can be derived straightforwardly from the instantiation directives. A valid composition is possible if all conflicts can be resolved, i.e. all classes in the resulting metamodel, and properties and operations of merged classes, need to have unique names.

Transformations

The produced basic transformations comprise name mappings and specifications for default objects and values which ensure that conformance is re-established when properties with a non-zero lower bound multiplicity are added (either in code blocks or as a consequence of class merging). To achieve this, the name mappings are used to perform three operations on the model level. First, name mappings ensure that a source model object is made an instance of the correct class in the target metamodel when its class has been renamed. Second, name mappings make sure that the slots in an object bind to the correct class properties. Third, objects are merged when several class name mappings have the same target name.

6.2.3 Design Decisions and Limitations

The framework for derived migration of models utilises the template instantiation directives for deriving transformations that migrate or compose existing models. The framework and its use can be seen as an operator-based approach [75]. This comes from the fact that the adaptation and composition directives are reflected directly by co-evolution transformations on the model level. However, with respect to other operator-based approaches available, we take things a step further since templates may evolve in hierarchies. Therefore we calculate collapsed forms of the transformations that reflect how several directives may be used on the same elements along branches in a template hierarchy. The directives yield non-breaking and breaking resolvable changes [72] on the model level when migration follows a single branch (which is the situation where these terms make sense). The directives can also be classified as model-preserving and safely model-migrating [64] (if we consider them as coupled operators). Only evolution resulting from applying the renaming directives can be reversed by the mechanism (by using renaming again).

Even though model conformance can be re-established during migration, it is typically required to manually revise generated default objects and values. The instantiation directives and generated transformations (coupled changes) are metamodel-specific and model-independent [71], which means that their application is specific to the contained metamodel of the template being instantiated, and that the application of the directives reflects equally upon all existing models of the metamodel.

The main drawback of operator-based approaches has been identified as the need to modify (i.e. adapt) metamodels using dedicated tools [18]. Changes to a metamodel that is e.g. performed manually will not be reflected in the corresponding model co-evolution operations. This is also the case with metamodel templates, which means that all changes to a metamodel of a template have to be performed using a combination of the available instantiation directives. Adapting a metamodel manually or using another tool means that type-safety can no longer be guaranteed and that corresponding transformations for migration and composition of models are not generated for the particular adaptations.

The available directives do not cover all kinds of metamodel evolution scenarios. The only way metamodel templates can be used to cover more evolution scenarios is to add more directives. To support such directives we need to find a solution to how automatic model migration can still be achieved.

Destructive directives induce difficulties when it comes to automatic model migration. For instance, deleting a class means that model objects of this class have to be discarded which will change the meaning and soundness of the models. Similarly, deleting attributes or references means that values and objects of these attributes and references, respectively, are discarded or dereferenced. When a property is moved to another class (which is not a superclass of the current class) it would be possible to transfer the attribute values to objects of the class to which the property has been moved or create new (reference) links to ensure that model conformance is preserved. This requires a way of mapping a removed property to an added property, e.g. as happening when a property is moved. However, when destructive directives are used it is not possible to infer sound models automatically, i.e. we get breaking and unresolvable changes which require interaction by the user.

Paper VI discusses the approach in greater details.

6.3 Artefact C: Framework for Metamodel Composition and Adaptation with Model Migration

We have created a framework for analysis of metamodel composition and adaptation that provides means for updating/migrating and composing the existing models of the metamodels given as input to the framework.

6.3.1 Overview

The framework analyses whether a user-specified set of composition and adaptation operations applied on a set of metamodels yields a well-formed resulting metamodel(s) and, if this is the case, calculates a set of effects which describe the minimal number of changes that must be applied to existing models of these metamodels in order to re-establish model conformance. MDE promotes the use of several languages simultaneously for modelling of a software system, e.g. using a multi-view approach where different concerns or aspects are modelled independently from each other. Other approaches in MDE focus on constructing metamodels from smaller building blocks and extending them with new features. In many cases, this means that several metamodels need to be composed (simultaneously).

The framework handles situations that deal with multiple metamodels and generates effects describing how to compose or migrate existing models, i.e. models conforming to each metamodel used in the composition (if such models are available); each set of source models need to be updated differently to re-establish conformance. The effects generated comprise name mappings and object/value creation specifications. The framework supports composing and adapting an arbitrary number of metamodels according to an *adaptation strategy.* In this case, the term *adaptation* also covers composition. A strategy contains a list of operations whose application is analysed on the metamodels provided as input. The framework supports 16 operations, including *class merging*, *class overriding* and *class interfacing.* It also supports diverse operations for refactoring purposes. The operations are formalised in both Paper IV and Research Report I. The framework is formalised as a deductive system for judgements. It operates in two phases: the metamodel analysis and adaptation/composition phase and the model migration/composition phase. The phases preserve metamodel consistency and model conformance, respectively, as discussed in [63].

6.3.2 Metamodel Analysis and Adaptation/Composition

In the first phase, the operations of the adaptation strategy are applied sequentially on a number of metamodels. The analysis checks whether the adaptation strategy is sound with respect to the metamodels, i.e. that it can be applied successfully and that the resulting metamodel(s) is well-formed. It also resolves name conflicts.

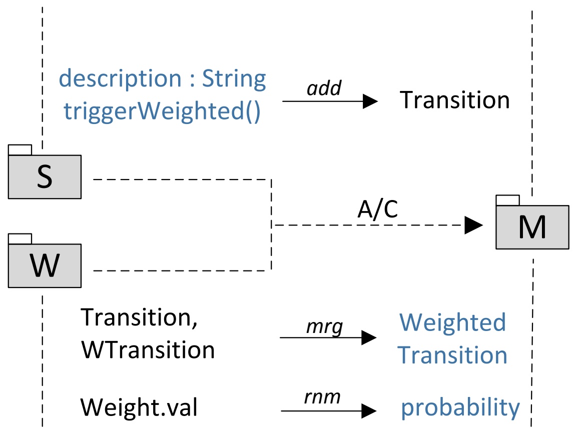


Figure 6.21: Adapting the metamodel for modelling of state machines

The result of the analysis is a set of adapted (and composed) metamodels and accumulated effects which are used in the model migration and composition phase. The effects specify elements that need to be renamed to resolve conflicts and specifications of objects and values that need to be created to ensure conformance.

Figure 6.21 illustrates how the metamodel for state machines can be extended with support for weighted transitions. The classes are the same as in Figure 6.5. An adaptation strategy supporting this case is given in Figure 6.22.

$$\begin{aligned}\Phi = \; & addProp(Transition, \langle String, description, (0,1)\rangle) \cdot \\ & addOp(Transition, \langle \epsilon, triggerWeighted, \epsilon\rangle \cdot \\ & merge(Transition, WTransition) \cdot \\ & renameProp(Weight, val, probability)\end{aligned}$$

Figure 6.22: Adaptation strategy for supporting weighted transitions

The strategy specifies that a property named description of type String and an operation named triggerWeighted with no return type or parameters should be added to the Transition class. A merge operation is then used to specify that the Transition class and WTransition classes should be merged. Finally, the val property of the Weight class is renamed to probability.

6.3.3 Model Migration/Composition

The effects generated by the analysis specify the changes that need to be performed on existing models for these models to be conformant with the resulting metamodel(s). In Paper IV we outline a function that transforms any existing model of any metamodel provided as input to the framework (assuming that the adaptation strategy was applied successfully). The function does this by going through each operation of the adaptation strategy and updating the model respectively. This includes querying the generated effects. The function handles operations used for metamodel composition as well, e.g. merging of classes, and ensures that existing models of the input metamodels are updated to be conformant with the resulting metamodel. Put differently, the function does not compose two or more existing models, but ensure that each model is migrated to a valid instance of the resulting metamodel. However, the adaptation strategy and generated effects may also be used as input to a function for model composition that composes a number of user-specified models.

Figure 6.23 illustrates the two ways metamodel composition, in the form of class merging, can be addressed at the model level, i.e. by migrating the existing models to conformant versions of the composed metamodel or by composing the existing models (as selected by the user) to a model of the composed metamodel.

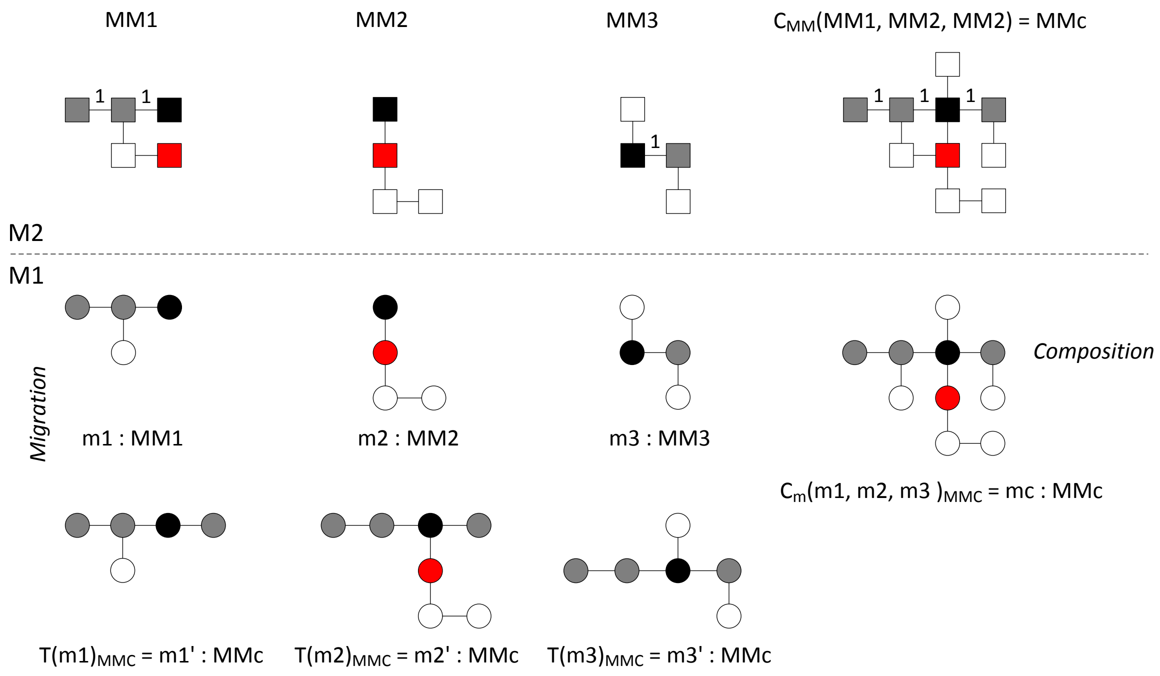


Figure 6.23: The two ways merging of classes can be addressed on the model level

The figure shows three metamodels that are composed by class merging ($C_{MM}(...)$), i.e. the three black classes and the two red classes in the metamodels are merged. Some of the references have a non-zero lower bound for their multiplicities. i.e. the lower bound is 1. This means that (at least) one object of the related grey class needs to be created. Models of MM1, MM2 and MM3 are shown. The models are composed ($C_m(...)$) to constitute a model of MMc. They are also migrated ($T(...)$) into versions that are conformant with MMc.

$$\begin{aligned}\sigma = \ & [Transition \mapsto_c WeightedTransition] + \\ & [WTransition \mapsto_c WeightedTransition] + \\ & [val \overset{Weight}{\mapsto}_p probability] \\ \delta = \ & \langle\langle WeightedTransition, weight, 1\rangle\rangle\end{aligned}$$

Figure 6.24: Effects generated by the analysis

The generated effects produced by the analysis for the running example are given in Figure 6.24[5]. The name mappings (σ) reflect the *merge* and *renameProp* operations. A value specification is generated (in δ) which states that a default object (of type Weight) should be generated for the weight reference (attribute) in the WeightedTransition class. This is required since the lower multiplicity of this reference is 1.

6.3.4 Multiple Metamodels

The framework supports composition and adaptation of an arbitrary number of metamodels simultaneously. In other words, an adaptation strategy may contain operations pertaining to any number of metamodels. The advantages of this is particularly evident on the model level. The available model migration approaches typically support migration between two versions of a metamodel and do not support composition of metamodels directly. A common approach in MDE is to use several languages to model a system. This means that different concerns or aspects (views) can be modelled separately and later be composed to create a composed model for the whole system. The composed model conforms to a composed metamodel, and the production of the composed model reflects the adaptation and composition operations applied on the various metamodels. The framework supports this multi-language approach efficiently, since adaptation and composition of an arbitrary number of metamodels and models are treated simultaneously. Also in situations where a metamodel is extended with additional features, as defined in several metamodel fragments/patterns our approach may be useful, i.e. if there exist models of the metamodel fragments that need to be composed with the models of the metamodel being extended.

6.3.5 Formalisation

We have formalised the operation of the framework. Not many approaches for (meta)model composition and adaptation, or model migration formalise their operations. The authors of [88] argue that the impact assessment and the adaptation semantics are typically mixed in current approaches for co-evolution, and that formalising migration approaches improves the insight and information accessible to designers which may improve the design and realisation of tools.

The formalisation we have used improves readability of the adaptation and composition operations and makes it easier to understand the premises and the result of an operation, i.e. the mechanics of the framework is made clearer. Thus, it provides an optimisation space that is easier to reason about than an implementation. For instance, it is easy to extend the framework with additional operations, which are added as new rules. And importantly, we have used the formalisation

[5]We assume that WeightedTransition is the name generated by the framework.

to prove that the framework operates correctly i.e. with respect to re-establishing model conformance.

6.3.6 Design Decisions and Limitations

The framework allows specifying an adaptation strategy that when applied composes and adapts metamodels. The model migration part of the framework is a hybrid between an operator-based and change recording approach. This is because we create a record of the changes (effects) which are later used in model co-evolution transformations to migrate (or compose) the models. Together the operations and the effects (i.e. the name mappings) can be seen as a kind of difference model [70] which describes all metamodel changes. The format of adaptation strategies is generic, which means that an adaptation strategy may be shared by different modelling frameworks.

The operations supported yield non-breaking and breaking resolvable changes [72]. They can also be classified as model-preserving and safely model-migrating [64] (if we consider them as coupled operators). Evolution resulting from applying the renaming and pull/push operations and can be reversed by applying inverse operations. There exists a parallel dependency [70] between some of the operations, e.g. a property or operation can not be added to a new class before the class has been created. The dependencies can be handled straightforwardly by an appropriate manual sequencing of the operations. Since the sequence of the operations is reflected in the model migration phase (i.e. the adaptation strategy drives the model migration), there are no unresolved dependencies that occur during migration. This is not the case for difference-based model migration approaches where dependencies need to be resolved. That is, difference models do not contain information on the sequence of the applied metamodel changes. The operations and generated effects (viewed as coupled changes) are metamodel-specific and model-independent [71].

As with the metamodel template mechanism, the framework does not support destructive operations. This can easily be added by defining new rules (which potentially build on the existing rules). Operations for deleting and moving elements induce the same difficulties on the model level with respect to automatic model migration, i.e. discarding objects and values will not preserve the intended meaning of the models which implies that manual intervention is required. Moving properties and operations up and down in class hierarchies is partially supported. That is, we support versions of such operations that preserve model conformance.

As similarly argued with metamodel templates, we do not support merging of attributes, references and operations since such elements may have different semantics even though their names (and parameters) are identical. Adding operations to support this is not difficult should such be required. However, operations for merging of attributes and references may impair the ability to automatically compose models (required after metamodel composition) in certain situations. For instance, let us assume that two references from two different metamodels are merged. The first reference has a multiplicity of `0..n`, whereas the other has a multiplicity of `0..m`. We assume that references can be merged even though their multiplicities are not identical. In this situation, the merged reference gets the most general multiplicity of the two references. The problem occurs when `n` and `m` are both specific integers (i.e. not `*`). In this situation, due to model composition, the total number of objects intended to be associated/linked by the merged reference may exceed the upper bound of its multiplicity. This means that objects have to be discarded in order to produce a model that conforms to the composite metamodel. Deciding what objects to discard can not be performed automatically. That said, it can be difficult to justify merging of references that have different multiplicities, since these references are likely to have a different semantics and are therefore not suitable for being merged. Also, the most common upper bounds are `1` and `*`. Therefore, it only makes sense (semantically) to support merging of references as long as 1) the types of the references (classes) and the classes containing the references are evaluated to be semantically equal and merged respectively, and 2) the multiplicities are compatible, i.e. when the lower bounds are identical and one of the upper bounds is `*`, or when the lower bounds are identical and both upper bounds are either `*` or `1`. The latter case makes sense since the referenced objects in the source models (either zero or one in each model) are likely to be composed as well. Similar reasoning applies for merging of attributes.

The work of [63] closely resembles our work. The approach is based on using coupled transactions (coupled changes), which means that there is a corresponding model migration segment being generated for each metamodel adaptation. What we do instead is to generate effects which are accumulated throughout the analysis. We then use the effects in addition to the adaptation strategy to update the models. As discussed, in situations where there are more than one source metamodel there are two ways to reuse/update the existing models in order to ensure model conformance. Both ways of updating the models can be achieved from the generated effects and the adaptation strategy. The approach discussed in [63] only supports the first situation, i.e. migration (corresponding to adaptation of one metamodel) and not composition (corresponding to composition of metamodels) of the existing models. Model migration and composition can be described in terms of two distinct algorithms. The algoritm for model migration is sketched in Paper IV.

Metamodel composition can by [63] (and other approaches) be described as an adaptation in terms of using several operations for adding new classes and class attributes, and operations for adding relationships between these classes. Also, difference-based approaches can detect such additions and generate transformations for updating existing models of the source metamodel. However, with Artefact C, it is possible to compose pre-made metamodels directly, and importantly, we can differentiate and migrate/compose models conforming to all of the source metamodels. This is not possible with any migration approach that we are aware of.

Paper IV explains in detail how the analysis works and how the effects are generated.

6.4 Artefact D: A Framework for Non-Intrusive Integration of Operational Semantics

Composing metamodels will in most cases impact existing models and model management tools which are rendered invalid. We have defined a framework that allows integrating the operational semantics of metamodels more or less non-intrusively. This is achieved by creating a set of mappings between elements of different metamodels.

6.4.1 Overview

Composing the structures of metamodels usually means that existing models and tools are impacted. Hence, there is a clear incentive for enabling composition of metamodels without explicitly combining their structures. One way of achieving this is to utilise proxy classes. A proxy class in one metamodel conceptually represents a class in another metamodel. This is achieved by creating mappings between the two classes and their contents. The mappings comprise a unification point between the two metamodels. Two metamodels may be unified with one or more unification points as described in a unification model. We refer to the mappings as *M2-mappings* as they connect elements on the M2 layer of the MOF metamodelling architecture.

A class in one metamodel may be a proxy for a class of another metamodel if all the attributes and operations of the proxy class can be mapped to attributes and operations of the target class, i.e. the classes are conceptually equal from the perspective of the proxy. (The target class may contain additional attributes, references and operations.) At runtime (i.e. when the operational semantics is executed), an object of a proxy class therefore (partially) represents an object of the target class. Or more specifically, the attributes and operations of the proxy class represent a selection of the attributes and operations of the target class. It is possible for a proxy to represent all attributes and operations of a target class. As before, we use Kermeta in the examples (the implementation works on Ecore models (metamodels)/Java, see Paper V).

6.4.2 Unification at M2

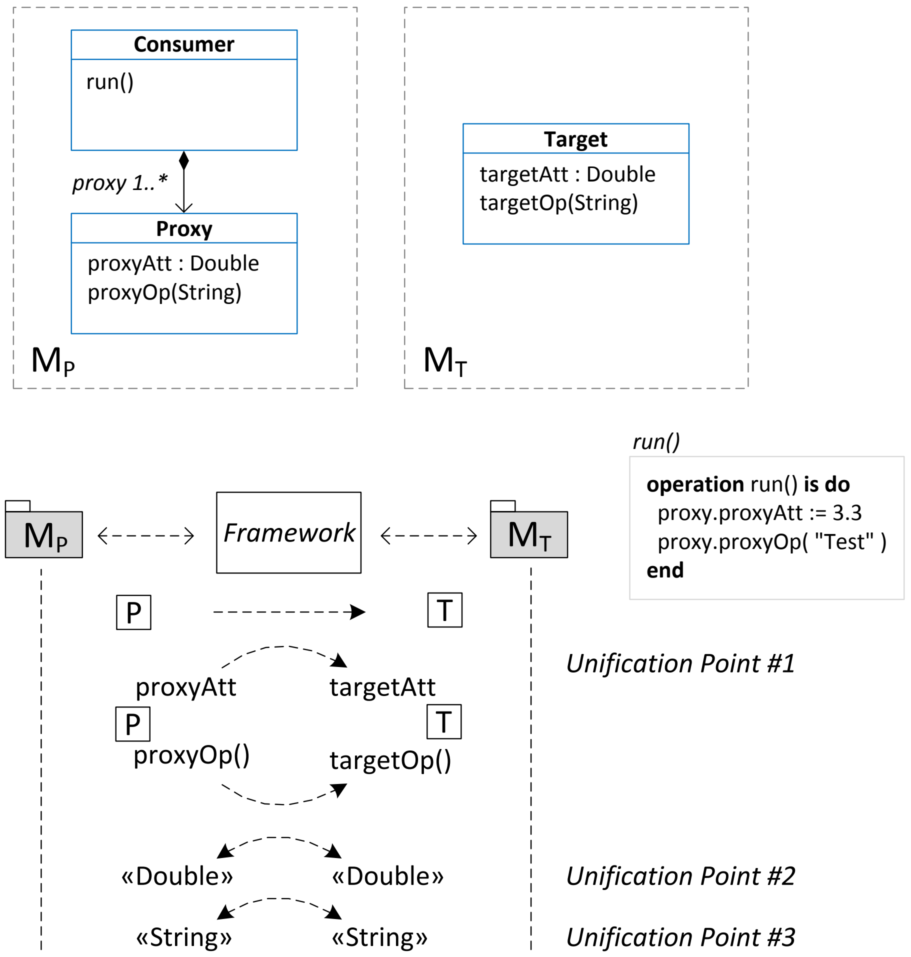
Figure 6.25 shows the general scheme of how a proxy class represents a class of another metamodel.

```
// MMA
class Consumer {
  attribute proxy : Proxy[1..*]

  operation run() is do
    proxy.proxyAtt := 3.3
    proxy.proxyOp( "Test" )
  end
}

class Proxy {
  attribute proxyAtt : Double
  operation proxyOp( val : String ) is do ... end
}

// MMB
class Target {
  attribute targetAtt : Double
  attribute otherAtt : Integer

  operation targetOp( name : String ) is do ... end
  operation otherOp() is do ... end
}
```

Figure 6.25: Proxy and target classes

MMA has a class named Proxy that consists of an attribute proxyAtt and an operation proxyOp. The operation run() in the Consumer class merely sets a value for the attribute and invokes the operation. MMB contains a class named Target with an attribute named targetAtt and an operation named targetOp. It also contains an additional attribute and operation which are not relevant for the integration.

Figure 6.26 shows an overview of how the operational semantics of the two metamodels are integrated. Specifically, the proxyAtt attribute of Proxy is mapped to represent the targetAtt of Target, and the proxyOp(...) operation of the Proxy class is mapped to represent the targetOp(...) operation of the Target class. Both attributes are typed with Double. A mapping reflects that both Double types are identical. Similarly, both operations have a parameter of type String which yields a mapping as well. The four mappings yield three *unification points*.



```
operation run() is do
  proxy.proxyAtt := 3.3
  proxy.proxyOp( "Test" )
end
```

Figure 6.26: Integration of the operational semantics

A unification point specifies (partial) structural equivalence between two classes or types. Unification of the Proxy and Target classes gives one asymmetric unification point (partial equivalence), whereas the unifications of the Double (x2) and String (x2) types yield two symmetric unification points (full equivalence). A symmetric unification point indicates that the classes/types unified are identical type-wise. For example, both the Double types in the two metamodels represent floating point numbers (which is trivial as both metamodels are defined using the same metamodelling language).

In the example, it is easy to see that the mappings between the Proxy and Target classes give a valid unification type-wise. Specifically, the attribute proxyAtt and targetAtt are both of type Double, whereas the operations have no return type and a parameter of type String.

6.4.3 Runtime

Two metamodels and a collection of unification points describing their structural integration resemble what is known as a *multimetamodel* [110]: $\mathscr{M} = (M_1, M_2, M_{12}, r_1, r_2)$, where a pair $x_1 \in M_1, x_2 \in M_2$ is the same (i.e. representing a structural overlap) if $x \in M_{12}, r_1(x) = x_1, r_2(x) = x_2$. M_{12} contains the common concepts between the metamodels, whereas r_1 and r_2 are graph mappings.

The mappings represent a conceptual composition of the two metamodels. During execution of the semantics, i.e. when the run() operation is invoked, the framework redirects the assignment of the proxyAtt attribute in the run() operation to the targetAtt attribute. That is, the targetAtt attribute, and not the proxyAtt attribute, is assigned the value 3.3. Furthermore, the invocation of proxyOp(...) in run() is resolved as an invocation of the targetOp(...) operation. What we achieve is that the operational semantics of the metamodels are integrated in an almost non-intrusive manner, i.e. the metamodel structure has not been changed in order to integrate the semantics.

6.4.4 Linking at M1

There will typically be many models/programs of a metamodel. Moreover, the models may contain several objects of the same classes.

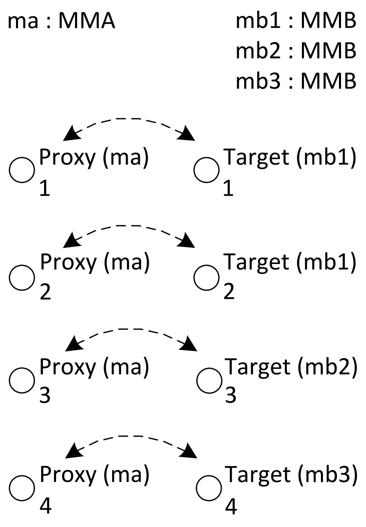


Figure 6.27: Linking model for the example

At runtime the correct models and objects need to be linked. This is achieved by creating *M1-mappings* between objects. A set of M1-mappings yields a linking model. Figure 6.27 illustrates such a model. The model maps four objects of the Proxy class in the model ma to four objects of the Target class distributed across the models mb1, mb2 and mb3.

6.4.5 Design Decisions and Limitations

The framework for integration of operational semantics makes it possible to execute the operational semantics of different metamodels in concert, by establishing mappings between metamodel structures and model objects, and adding glue/bridging code that either set the values of attributes or invoke operations. The main advantage of using proxies is that the code in the source metamodel has implicitly access to structure of the classes in the target metamodel (since this is the same structure as that defined for the proxy classes). This means that we do not need to import the target metamodel and thereby (mostly) avoid changing the metamodel in ways that induce impacts on other artefacts. Every non-abstract class in a metamodel can be a proxy class. A proxy class can also be added e.g. by declaring it as a subclass of an existing class. For the latter case, this means that the metamodel structure has been extended/changed. However, existing models are still valid after the subclass has been added. Tools are also compatible with the new metamodel version unless a model contains instances of the subclass (reflective model editors will automatically support the subclass). Moreover, creating an instance of a proxy class is only relevant for integration purposes, i.e. as a proxy represents a concept of another metamodel.

Defining unification points resembles how signature matching works for a signature-based model composition approach. The signature type for an asymmetric unification point comprises all the elements of the proxy class, i.e. these elements must be matched by corresponding elements in the target class. An element matches another based on a qualified choice by the user (i.e. elements are matched manually). The name of the elements do not need to match; only the type. For symmetric unification points all elements of both classes/types in question must match. The unification model can also be seen as a weaving model [88], which enables the creation of *equality* links for the purpose of *interoperability* and *data integration* [22], i.e. we are able to align models.

A difficulty with establishing mappings between metamodels is that elements in the different metamodels, which seem like good candidates for being unified, may express semantically different concepts. This means that even though all the elements of a proxy class may be mapped to the structure of a target class, it is difficult to assert that the proxy and target classes have the same semantics (or at least may be unified in a way that makes sense semantically). In the general case, it is very difficult to infer the semantics of concepts from their syntactic form [20] which means that the semantics of the metamodels need to be evaluated before establishing mappings [48]. This applies to all approaches that compose metamodels. The potential issues resulting from semantic incompatibility first emerge when code is executed, since the execution of the code reifies the intended operational semantics of a metamodel. For our approach, the metamodel classes that relate to the proxy class have associated model code (as defined in corresponding Java classes)[6] that expects the values it processes to reflect the pre-decided semantics of the proxy class. If the semantics of the target class, i.e. the model code of this class, differs from the expected semantics (i.e. that of the proxy class), problems may occur at

[6]In EMF, the abstract syntax is defined in Ecore models (metamodels) and operational semantics in associated model code.

runtime.

For the metamodel template mechanism we allow adapting the code by overriding operations, which means that problems related to semantic differences may be addressed. Similarly, the model code of classes relating to the proxy class may be adapted by updating this manually. The main premise for our approach is that metamodels are changed as little as possible in the process of integrating the operational semantics. Changing the model code means that metamodels, or more specifically their model code, are changed in an intrusive manner. However, changing the content of Java methods do not impact artefacts in the ecosystem with respect to structural compatibility. This is because the definitions of artefacts relate to the structural properties of a metamodel and not its operational semantics. These structural relationships (like *conformsTo*, *domainConformsTo*, *dependsOn*, etc. [3]) still hold regardless of how the code of methods is changed.

A related challenge when creating mappings between metamodels is that a concept in one metamodel may be represented differently in another metamodel. For instance, a concept represented by a class in one metamodel may be represented by several classes in another metamodel. And, an attribute (representing an implicit/hidden concept) in one metamodel may be represented by a class in another metamodel, etc. The current version of the framework only allows one-to-one mappings between elements (of the same kind, e.g. between two attributes). In theory, it would be possible to support one-to-many mappings, where the elements of a proxy class are mapped to elements distributed between several target classes. Mapping attributes or operations of the proxy class to attributes or operations of several classes in the target metamodel would not hinder a correct lookup at runtime.

One way of ensuring that one-to-one mappings are sufficient for mapping the structure of metamodels is to first utilise an ontology-based approach, e.g. by lifting the metamodels to ontologies [43]. The purpose of this operation is to make implicit concepts in the metamodels explicit (e.g. concepts represented by attributes) and improve the (meta)model alignment. This means that the user only has to specify one-to-one mappings between the ontologies. However, the one-to-one mappings between the ontologies still need to be transformed into mappings between metamodel structures which at least requires support for one-to-many mappings by the framework. Alternatively, a metamodel may be refactored, but this results in severe impacts on other artefacts.

A concept may also be expressed at different abstraction levels in different metamodels, which makes it difficult to establish mappings between the elements representing the concept. This can not be addressed without changing one or both of the metamodels. However, if this is the case, it is questionable whether the operational semantics of the two metamodels are ideal candidates for being integrated.

As have been illustrated, integrating the operational semantic of metamodels is not always straightforward, but the approach has several advantages in addition to reducing the impacts on artefacts, e.g. metamodels and models are kept autonomous and synchronised without the need to explicitly compose their structures, which in turn means that there is no entanglement of concepts. This means that concerns and aspects can be modelled in separate views and later be integrated merely by creating mappings between the metamodels and models.

The unification model expresses structural equivalence between elements of different metamodels, i.e. it is a type of weaving model that only allows defining equality relationships between elements. Of this follows that a proxy class overrides its target class. Hence, the actual composition of metamodels is in terms of relationships to the proxy class, as expressed in the source metamodel. The question is whether the unification model is expressive enough to perform complex compositions. A weaving model may utilise other types of relationships, like merge. However, it is not clear how e.g. merge relationships between elements can be realised in the code that integrates the operational semantics of the metamodels.

In Paper V we discuss how the framework allows integrating the semantics of a GPL metamodel with the semantics of a metamodel for modelling of state machines. The purpose is to support modelling of objects' behaviour in terms of state machine models. The paper also discusses how unification models can be verified to ensure that all unification points are valid.

6.5 Artefact E: A Theory for Realising Metamodel Types

There has not been much work on typing models or metamodels. We have identified class nesting as a promising mechanism for realising metamodel types. By enclosing a metamodel within a class, we are able to utilise established object-oriented mechanisms on the metamodel layer.

6.5.1 Overview

A metamodel comprises a number of classes. Each non-abstract class defines a type. The contents of the class, i.e. attributes, references and operations, define the properties of the type; this includes contents inherited from one or more superclasses. In other words, the class defines the properties of its instances (model objects). In object-oriented programming languages, a class may also contain inner classes. If an inner class is non-static, it contributes to defining the type of the outer class.

The underlying idea of our approach is to use class nesting for defining metamodel types. Specifically, an outer class contains an enclosed metamodel, comprising a set of non-static inner classes, that may act as a type for the enclosed metamodel. Both the outer and inner classes contribute to the definition of the type, and we refer to these classes collectively as a *metamodel type*. The outer and inner classes can be subtyped to specialise a metamodel, which supports substitutability and polymorphism.

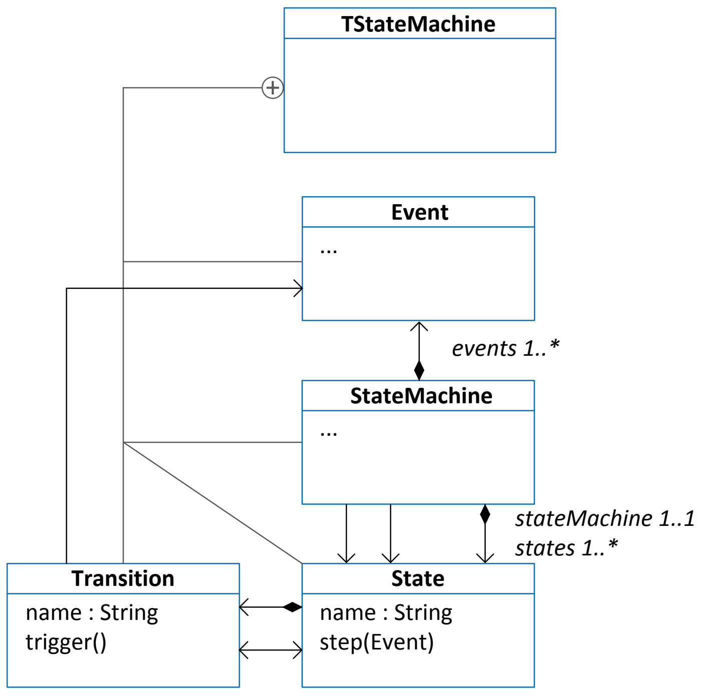


Figure 6.28: A metamodel type for modelling of state machines

Figure 6.28 shows how class nesting can be used to create a type for the state machine metamodel. The metamodel is enclosed by the TStateMachine class, i.e. the TStateMachine class acts as a type for the enclosed metamodel. As can be seen, four classes are enclosed in the TStateMachine class. Figure 6.29 gives the implementation of the metamodel type. The classes of the metamodel are accessed using an instance of the TStateMachine class. This is required e.g. when creating a model conforming to the enclosed metamodel.

Figure 6.30 illustrates how a model can be created programmatically. The model is stored by the sm variable which is typed with the top node/root class of the metamodel (all model objects are transitively accessible from an object of the top node class).

```
package state_machine;

class TStateMachine {
  class StateMachine {
    attribute states : State[1..*]#stateMachine
    attribute events : Event[1..*]
    reference currentState : State[1..1]
    reference initialState : State[1..1]
    operation run() is do ... end
  }
  class State {
    reference stateMachine : StateMachine[1..1]#states
    reference incoming : Transition[0..*]
    attribute outgoing : Transition[0..*]#source
    operation step( event : Event ) is do
      ...
    end
  }
  class Transition {
    attribute name : String
    reference event : Event[1..1]
    reference source : State[1..1]#outgoing
    reference target : State[1..1]
    operation trigger() is do
      source.stateMachine.currentState = target
    end
  }
  class Event { ... }
}
```

Figure 6.29: Implementation of the metamodel type for modelling of state machines

```
operation createModel() : TStateMachine.StateMachine is do
  // Creates an instance of the metamodel type
  var tsm : TStateMachine init TStateMachine.new
  // Creates an instance of the top node class
  var sm : TStateMachine.StateMachine init tsm.StateMachine.new
  // Creates a state and adds it to the state machine
  sm.states.add( tsm.State.new )
  ...
  // Assigns the model as the result from the operation
  result := sm
end
```

Figure 6.30: Creating a state machine model

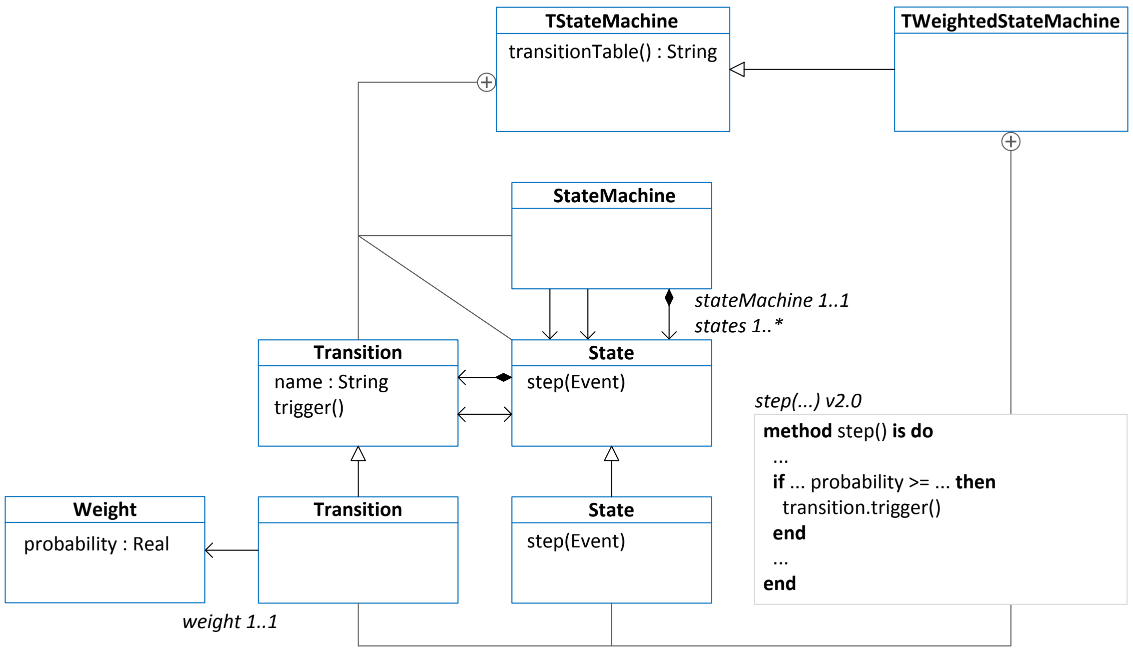
Figure 6.31 illustrates how the parameter of an operation (e.g. implementing a model management operation) can be typed with the StateMachine class from the metamodel type. The TStateMachine type is a reference type. The operation can process models conforming to all metamodels of metamodel types that subtype this reference type.

```
operation transitionTable( sm : TStateMachine.StateMachine )
  : String is do
  ...
  sm.states.each{ s | ... }
  ...
end
```

Figure 6.31: Typing a parameter with the StateMachine class of the metamodel type

6.5.2 Specialisation of a Metamodel

Both the outer class and the inner classes of a metamodel type may be specialised. Let us see how the TStateMachine class and the enclosed metamodel classes may be specialised to support modelling of state machines with weighted transitions. Figure 6.32 illustrates how this is achieved. First, a class for the specialised metamodel type named TWeightedStateMachine is created. The class extends TStateMachine. Second, the classes Transition and State are specialised by creating new classes in TWeightedStateMachine that extend the respective classes in TStateMachine. The Transition class is extended with a new attribute (containment reference) named weight of type Weight, whereas the specialised State class overrides the inherited step(...) operation with new code for evaluating the value of the probability attribute in the added Weight class. In Kermeta, overridden operations are specified using method instead of operation. The Event class is omitted from the figure.



```
method step() is do
 ...
  if ... probability >= ... then
   transition.trigger()
 end
 ...
end
```

Figure 6.32: Specialising the metamodel type to support weighted transitions

Figure 6.33 gives the implementation of the specialised metamodel type. The code supports having state machine models with both regular and weighted transitions. This time the step(...) operation contains the code for evaluating the weight associated with a transition.

```
package weighted_state_machine;
require "state_machine.kmt"

class TWeightedStateMachine inherits TStateMachine {
  class State inherits TStateMachine.State {
    // Overridden
    method step( event : Event ) is do
      var transition : Transition init
        outgoing.select{ t | t.event.equals( event ) }

      if transition.isKindOf( Transition ) then // Weighted transition
        // generates a random number between 0 and 1
        var randomNumber : Real init ...
        if transition.asType( Transition ).weight.probability
          >= randomNumber then
          transition.trigger()
        end
      else // Regular transition
        transition.trigger()
      end
    end
  }
  class Transition inherits TStateMachine.Transition {
    attribute weight : Weight[1..1]
  }
  class Weight {
    attribute probability : Real
  }
}
```

Figure 6.33: A metamodel type for weighted state machines

Figure 6.34 illustrates how the metamodel for modelling of state machines is specialised (adapted) and what classes that are instantiated in a model to utilise weighted transitions. The model editor first creates an instance of the specialised outer class (as illustrated in Figure 6.30) and the user then instantiates the specialised State (S) and Transition (T) classes (M refers to the StateMachine class). It is possible to have state machine models containing both kinds of transitions. A regular state is selected by instantiating the original Transition class. Transformation of existing state machine models, with the purpose of including weighted transitions, is straightforward by transforming all objects of the State class to instances of the specialised State class and a selection (or all) of the Transition objects to instances of the specialised Transition class. This includes creating an object of the Weight class for each weighted transition which is required due to the lower multiplicity of the weight attribute in the specialised Transition class. Objects of the Event class are omitted from the figure. A model comprises objects of the inner classes. The object of the outer class is not considered a part of the model.

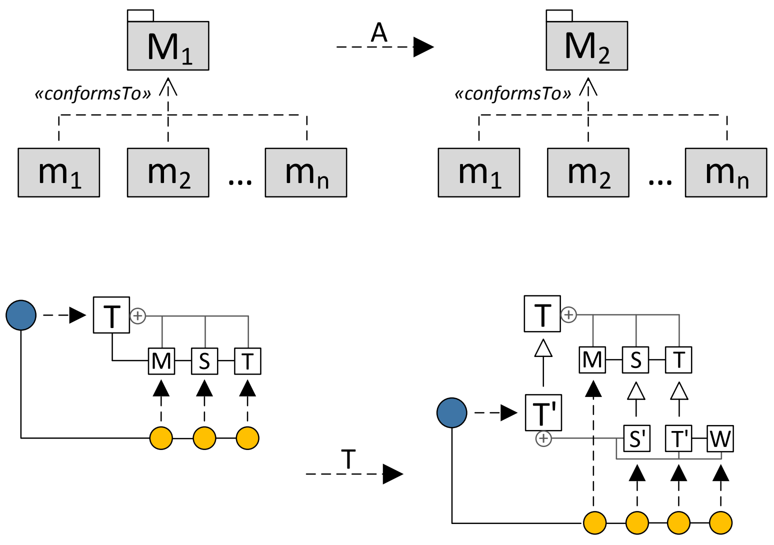


Figure 6.34: Transforming state machine models to utilise weighted transitions

Since specialisation of a metamodel is achieved using subtyping, we get additional polymorphism at the metamodel level. Specifically, a tool may take an instance of a class in a metamodel subtype as input and still operate correctly according to the classes and operations of the original type (supertype). Consequently, we are able to reuse both metamodels and tools.

6.5.3 Customising the Metamodel

So far the outer class has only enclosed the metamodel classes. The outer class may have additional attributes. The attribute values may determine certain characteristics of the enclosed metamodel. For instance, the operational semantics of several metamodel classes may be tuned by adjusting the value for the attributes. This means that the operational semantics of the metamodel can be customised at a global level. The customisation applies to all models of the metamodel, without the need to explicitly alter the models, i.e. changing values in objects of the metamodel classes.

In Paper III we also add a reference to the top node metamodel class (i.e. the StateMachine class), which means that tools and editors may reference the current loaded model on which they work on. By having such reference it is also possible to utilise operations in the outer class. The operations can be used by tools and editors to e.g. generate information about the loaded model. Figure 6.35 shows such an operation for generating the transition table.

```
package state_machine;

class TStateMachine {
  // References the currently loaded model
  reference sm : StateMachine[1..1]

  operation transitionTable() : String is do
    sm.states.each{ s | ... }
    ...
    result := ...
  end

  class StateMachine { ... }
  ...
}
```

Figure 6.35: Global operation for generating the transition table

6.5.4 Using Several Metamodel Types

The MOF metamodelling architecture supports multiple inheritance. This means that a class may have several superclasses. For metamodel types this is interesting because it allows defining a metamodel type using several supertypes. The overall type for a metamodel can thus be seen as a combination of several types.

```
class TSomeType inherits TStateMachine, TX, TY, TZ {
  class M inherits TStateMachine.StateMachine, TX.X {
    reference y : TY.Y[1..*]
    ...
  }
  ...
}
```

Figure 6.36: Defining a metamodel type using multiple inheritance

Figure 6.36 gives a metamodel type named TSomeType that inherits from four supertypes. The definitions of the classes in TSomeType (either new or redefined) combine the inherited elements from the supertypes either through inheritance or by adding references. The M class exemplifies this. Each supertype represents a specific aspect or concern of the enclosed metamodel. In other words, each supertype gives a specific perspective on the metamodel. As an example, there may be a tool that is typed with the StateMachine class of TStateMachine. The tool only considers model objects pertinent to describing a state machine, regardless of whether these objects are linked with other kinds of objects used to describe other aspects of the metamodel. What this means is that existing tools may be used for a large number of metamodels as a metamodel type can inherit from the types used to define the tools.

6.5.5 Design Decisions and Limitations

We have discussed how class nesting can be used for defining metamodel types. The outer class acts as a type for the enclosed metamodel. The input of an artefact can therefore be typed by the outer class and one of its inner classes (e.g. the root class), which ensures an intuitive way of type-safe reuse.

We have seen how multiple inheritance can be used to define a metamodel type that can be viewed/processed according to several concerns (supertypes). In such cases, the metamodel type represents a composition of structure for modelling of several concerns. Using supertypes this way is particularly powerful as tools defined relatively to the supertypes may be shared by diverse metamodel types, which is a goal in itself [93], and resembles how concepts impose requirements on a metamodel as discussed in [42]. Similarly, the supertypes impose requirements on the metamodels as imposed by tools (model management operations).

Type reflection can be implemented straightforwardly when it comes to metamodel types since all features of a metamodel type are defined in terms of inner classes and properties of these classes (and eventually as attributes in the outer class). A model object is an instance of an inner class. This means that metadata for the outer class can be accessed reflectively from any model object which in turn gives access to all the inner classes. This allows determining the metamodel type of the metamodel that the object is instantiated from. The purpose of supporting type reflection is e.g. for producing generic tools. Type inference is related to type reflection. It is not obvious how type inference can be beneficial when it comes to metamodel types. However, we believe it can be supported without too much difficulties.

The authors of [99] discuss how languages can be composed in terms of nesting the classes of one language within a single class of another language, which the authors also refer to as a class refinement. This notion corresponds well to how the outer class (representing a concept or concern) is refined by the enclosed metamodel classes. In principle, this means that metamodel types can be used in several levels, i.e. an enclosed class may itself contain enclosed classes. A possible way this could work is that the outermost class represents a metamodel type containing different metamodels/languages for modelling distinct concerns or aspects of a system, i.e. the various system views. That is, the outer class (system type) contains a set of metamodel types (view types) which in turn contains metamodels. Modelling a concern or aspect of a system then translates to first instantiating the outermost class and then instantiating the metamodel type for the desirable concern or aspect which gives access to the metamodel for modelling of this particular concern or aspect. In theory, it would also be possible to use additional levels of metamodel types to support modelling at lower abstraction levels, e.g. a metamodel class enclosing a number of classes. The new level refines the class on the level above. Using class nesting this way corresponds to a composition according to the authors of [99], i.e. a class is composed of several enclosed classes. The practicability of nesting classes in more than two levels is not clear. It also requires having models that reflect the nested structure of the nested metamodel classes. When we have only two levels of class nesting we do not need this if the purpose of the levels are to add structure for representing views containing enclosed metamodels. A model will still have a flat structure consisting only of objects of classes within the innermost level, i.e. of a metamodel. (We here assume that a model represents one view). Yet, structuring the classes in different levels can be utilised by tools and editors, e.g. an editor may present the levels in an intuitive manner which makes it easy to navigate and find the correct view.

The operational semantics of a metamodel may contain code for deleting model objects. Let us assume that a subtype introduces a reference with a non-zero lower bound multiplicity. A model may potentially be rendered non-conformant during execution if the aforementioned code deletes objects that are linked to by the reference. However, this is not a problem that is unique for metamodel types. The same problem can happen using subtyping in e.g. Kermeta.

It is important to ensure that (instances of) classes of different metamodel subtypes are not mixed. Mixing instances of classes from different metamodel types may induce type errors. Let us assume a metamodel type comprising a class X containing an operation opX(...) with a parameter typed with a class Y. In a metamodel subtype the operation in the X class is redefined to invoke a new operation opY() that has been added to the Y class. Invoking the operation opX(...) in the X class with an instance of the original Y class will induce a type error since this version of the Y class does not contain the opY() operation. The type error occurs since Y is used as a parameter for the operation opX(...), i.e. the parameter in the subtype is covariant with respect to the type of the original parameter. Together with polymorphism this redefinition is not type-safe.

However, if we use virtual operations and overriding, we do not get a type error as illustrated. This is because the instance provided as argument to the invocation of opX(...) will be used to determine what version of opX(...) that should be invoked. That is, if an instance of the original Y class is provided as argument, then the original opX(...) operation will be invoked.

MOF does not provide semantics for redefining or overriding operations[7]. The theory on metamodel types is based on the Java type system. Java supports overriding methods and allows parameters to be contravariant and the return type to be covariant. Kermeta also allows overriding operations. By allowing overriding operations, instead of redefinition, the issue of mixed subtypes is addressed. However, redefinition can also be supported. The outer class of a metamodel type is instantiated by an editor. The editor may be configured to always instantiate the most recent version/subtype of a class, which in turn avoids mixing instances of different classes in a class hierarchy. This configuration also avoids errors related to type casting, i.e. when an instance of a supertype is tried being type converted to a subtype. This situation may happen when a reference typed with a supertype contains both instances of the supertype and a subtype.

[7]http://www.omg.org/mof

Covariance and contravariance in the case of operation redefinition are discussed further in Paper III.

In Paper I we investigate how the nested classes of a metamodel type may have generic parameters. This allows parameterising a metamodel with additional classes in order to temporarily adapt the metamodel for a specific problem domain. The paper also briefly discusses type hierarchies and using virtual classes. Paper III discusses metamodel types in greater details.

Current metamodelling languages (e.g. Kermeta as used for the illustrations) do not support nesting of classes. The theory on metamodel types is based on how class nesting is defined and used according to the Java type system.

6.6 Prototype Tools and Frameworks

Artefact A, Artefact B, Artefact D and the Additional Artefact II have been validated by means of prototype implementations. The prototypes are based on EMF and Kermeta.

6.6.1 Prototype Tool for Artefact A

The implemented prototype tool is defined as a Kermeta pre-processor. It is built using the *Eclipse Textual Modeling Framework (XText)*[8]. The pre-processor is compatible with a subset of the Kermeta metamodelling language. This subset allows defining metamodels containing operational semantics. The pre-processor supports organising templates in hierarchies and supports all of the constructs/directives listed in Chapter 6. This includes the ability to define bi-directional relationships and subtyping relationships between classes. All directives preserve type-safety and support re-establishing model conformance automatically. Kermeta was chosen since operational semantics can be defined directly in class operations.

6.6.2 Prototype Framework for Artefact B

The prototype tool for Artefact B builds on the prototype for Artefact A. It comprises a framework for (semi-)automatic migration and model composition which is achieved by deriving transformations from the instantiation directives used. This includes the capability of chaining migrations, i.e. transformations are combined using higher-order transformations. The term *(semi-)automatic* migration is used since models that are intended to be migrated or composed need to be chosen explicitly by the user. The implemented transformation engine is limited to composition of two models (i.e. their root objects are merged). However, the generated transformations support simultaneous composition of an arbitrary number of models (and composition of non-root objects).

[8]http://www.eclipse.org/modeling/tmf

6.6.3 Prototype Framework for Artefact D

The framework realising Artefact D supports defining mappings between metamodels. The mappings are expressed as a unification model which for the prototype is expressed directly in code (an alternative representation could have been an XML file). The same applies for the linking model. The prototype is compatible with EMF. That is, the operational semantics (model code) of Ecore models can be integrated. The glue/bridging code is added to the model code in a subclass of the respective Java class. The prototype does not check that unification points are valid (i.e. type compatible). This can be implemented straightforwardly by checking that the respective source and target elements are of the same type.

6.6.4 Prototype Language for Additional Artefact II

We have implemented a metamodelling language that supports creating metamodels with behaviour defined in class operations. The constructs of the meta-metamodel resemble those of KM3 and Kermeta. In addition, there are constructs for exporting and importing structure, i.e. attributes and operation signatures. Pre- and post-conditions and invariants can be specified for operations and attributes, respectively.

6.6.5 Other Artefacts

We have not implemented Artefact C, only formalised it. Artefact E is based on class nesting. Class nesting is not supported by any metamodelling framework/language (that we are aware of). Adding such capability to an existing metamodelling framework probably takes quite some work, and we have therefore not focused on implementing this.

Chapter 7

Discussion

This chapter is organised in three sections. In the first section we take a broader view of the artefacts and discuss general concerns and overall characteristics of our solutions. In the second section we evaluate our work. This includes evaluating the artefacts according to proposed metrics found in the literature and an overall evaluation of the work with respect to the requirements stipulated in Chapter 5. In the final section we discuss recent related work.

7.1 General Discussion

There are certain general properties and characteristics that are desirable for the artefacts. Here we will discuss the most important ones. We will also briefly discuss the additional artefacts as documented in the research reports.

7.1.1 Verification of Consistency of Compositions

The most important concern when composing metamodels and models is that the resulting metamodel or model represents a well-formed and sound combination of structures/elements. This e.g. requires that elements are evaluated for (semantic) equality (with respect to the problem domain) known as matching, that consistency between viewpoints are maintained, that conflicts are handled properly, and that certain properties are preserved, e.g. model conformance between a metamodel and its meta-metamodel (resulting in a well-formed metamodel) and model conformance between existing models and a given metamodel. We will now discuss each of these aspects with respect to the artefacts.

Matching of Concepts

For Artefact A and Artefact C, the user specifies a number of directives or operations describing how two or more metamodels should be composed. In other words, we do not use any form of automatic matching of metamodels, e.g. by defining pointcuts that identify join points. This means there is a potential for *concept misidentification* and *concept misses* [29], i.e. classes describing different concepts are wrongly matched and integrated and concepts that are represented differently are not identified as valid matches, respectively.

Viewpoint Consistency

In [90], it is argued that building a composed metamodel of several metamodels describing distinct concerns (viewpoints) may be difficult because such action may introduce viewpoint inconsistencies, i.e. because the abstraction level and level of granularity of the viewpoints may be different and because they may have different semantics. As discussed, lifting of metamodels to ontologies [43] can alleviate such difficulties. It is also possible to use ontologies for guiding the composition of metamodels. For Artefact C, the rules may be extended to consider ontological knowledge, e.g. by matching concepts against a reference ontology. For instance, the class merging rule may check that both classes intended to be merged can be traced to the same (or similar) concept of the ontology. If this is the case, the two classes represent a valid match. Another challenge pertinent to verification of model viewpoint consistency is that different viewpoints may use different notations [51]. However, for metamodels this is of a lesser concern since most metamodels (i.e. class models) are created using a common metamodelling framework and meta-metamodel (e.g. EMOF/Ecore). The authors of [103] argue that if a multiview model conforms to its metamodel, then the views are mutually consistent. Hence, by ensuring model conformance of a composed model (with respect to a composed metamodel) we also achieve consistency between the views as expressed in the source models.

Artefact D allows integrating the operational semantics of viewpoints without explicitly composing their structures, which helps maintaining viewpoint consistency. The glue/bridging code may address some variations regarding differences in semantics and how concepts are represented. However, as discussed, operational semantics of metamodels that are too far apart (semantically) will be difficult to integrate successfully.

Conflict Handling

Conflicts can either be avoided in advance or detected and resolved when they occur [92]. For Artefact A potential conflicts have to be manually addressed by using renaming directives. This requires the directives to be sequenced correctly, e.g. the conflicting elements of two classes have to be resolved before the classes are merged. For Artefact C conflicts are resolved both manually, and detected and resolved automatically by generating default names for conflicting elements.

Model Conformance

The artefacts supporting metamodel composition ensure that model conformance is preserved or re-established both at the metamodel level and model level if a given set of directives or operations are applied successfully[1].

A related problem of verifying consistency following (meta)model composition is checking the consistency of the migrated models. This becomes more challenging as the size of models increases [65]. As discussed previously, Artefact B and Artefact C ensure that model conformance is established/maintained between a successfully migrated model and its metamodel. However, it may be required to manually revise the automatically generated default objects and values. And important aid in identifying the default objects and values is to inspect the generated transformations/effects; which means that revision can be addressed efficiently regardless of model size.

7.1.2 Extensibility and Scalability of the Artefacts

An important aspect of an approach is how well its tools and operations can be extended to address new requirements, and how well the approaches scale to support more complex and detailed problems. These properties directly affect the capabilities and applicability of an approach.

We have foreseen some possible extensions of the artefacts, i.e. we have already discussed how Artefact A and Arfefact C can be extended with additional directives/operations, and how Artefact D can be extended to support one-to-many mappings between metamodel elements. These extensions mean that a broader type of problems can be supported. We have argued why supporting destructive directives/operations means that type-safety and (semi-)automatic model migration can be difficult to support. However, it would be possible to have two different operation modes for the artefacts. The first mode for Artefact A/B would only provide directives that preserve type-safety with support for (semi-)automatic model migration. The second mode would provide additional directives for a wider range of usage scenarios, though type-safety and (semi-)automatic migration would not be guaranteed. The modes would be similar for Artefact C.

The authors of [108] argue that *filtering* of concepts (e.g. a class or property) is preferred instead of deleting these when metamodels are extended. The reason backing this statement is that metamodels do not have to be changed intrusively which will render existing models invalid. In [108], filtering is used to hide a concept that should not be available in an extended metamodel[2]. The approach relies on

[1]We do not address the specific case where model composition may result in an object being contained by more than one object.

[2]The authors use the term *metamodel extension* even though filtering and modification of concepts occur, which justifies the use of the term *adaptation*.

deriving an extended (virtual) metamodel based on an original (or previously extended) metamodel. Using filtering, the user is not confused by existing classes in the metamodel that are not relevant anymore (even though they still exist). In other words, the extended metamodel acts as a filtered projection of the original metamodel. When metamodels are extended/adapted concretely things are not as straightforward since obsolete elements make the metamodel more difficult to comprehend and relate to by artefacts. The open question is whether filtering by model editors/graphical syntaxes is still a better choice with respect to deleting elements which will introduce compatibility issues with existing models and tools. A related work is *metamodel pruning* where an *effective* metamodel is calculated, i.e. unnecessary elements are removed, based on a set of rules and options [109].

Artefact B reflects the directives supported by Artefact A, i.e. the directives that work on metamodels are reflected by transformations on models which means that any extension of Artefact B is dictated by extensions of Artefact A. Artefact E is based on using class nesting (in a similar way as supported by the Java type system). The only possible extension we have identified for metamodel types is to use generic parameters on the outer class, which is discussed in Paper I. This allows abstracting details of a metamodel and customising the metamodel for a specific usage.

Whether an artefact can be extended depends on what kind of extensions that are desirable. For Artefact C it is possible to add new operations straightforwardly due to the modularity of the rules (which can also be reflected in an implementation of the framework/rules). This includes the ability to generate additional effects during the analysis. Artefact A and Artefact D can be extended fairly straightforwardly. However, how easy these artefacts can be extended strongly depends on how these artefacts are implemented, i.e. this is concerned with the architecture of the implementation. For the prototypes of Artefact A and Artefact D this requires some work.

We have not discussed static semantics (in the form of OCL statements) for any of the artefacts. Supporting such is a natural extension of Artefact A and Artefact C. If OCL is used, then renaming of elements and merging of classes require the referenced elements of invariants, pre- and post-conditions to be updated to reflect the new names given/produced [56]. For instance, an OCL statement may reference a class whose name is renamed, and thus, the OCL statement is rendered invalid. Since we do not support merging of properties, conflicts between several invariants should not occur. Also, supporting addition of global constraints during composition is desirable. A global constraint involves elements from several metamodels [110]. Providing means for automatic checking of global constraints during model composition is also desirable.

The artefacts are designed so that they scale well for bigger problems. We have not tested the following statements by an empirical study, but based on logical reasoning on the design decisions for the artefacts we believe that the following is true. Artefact A can handle arbitrarily large metamodels and template hierarchies (only limited by the design of the implementation and computer resources). An arbitrary number of metamodels can be adapted and composed simultaneously. The supported metamodels have to be defined using a subset of Kermeta; the subset includes the important language concepts. We do not foresee any problems with supporting the full version of Kermeta. The scalability of Artefact B corresponds to that of Artefact A, i.e. migration of models according to template hierarchies of arbitrary sizes is supported. Also, the size of the models being migrated is not of importance. For Artefact C we allow simultaneous adaptation and composition of an arbitrary number of metamodels of arbitrary sizes. The metamodels have to utilise the most common structural elements of EMOF. We do not see any difficulties with supporting the remaining concepts of EMOF. Artefact D allows integrating the operational semantics between an arbitrary number of metamodels of arbitrary sizes. This can be achieved by using an arbitrary number of proxies and mappings. Artefact E supports metamodels of arbitrary sizes. It also allows using as many supertypes as desirable through multiple inheritance.

7.1.3 Reuse and Expressiveness

The template instantiation directives and operations, of Artefact A and Artefact C, respectively, for composition and adaptation of metamodels have to be applied specifically to one or more metamodels. This means that a sequence of instantiation directives or a sequence of operations can not be reused and applied on other metamodels without manually specifying the new elements the directives or operations should work on.

Migration is achieved (semi-)automatically based on the directives or operations used. The current versions of the artefacts do not support deriving migrations for recurring migration knowledge [63] (corresponding to reusable adaptations).

A study of two industrial metamodels showed that 31% and 48%, respectively, of all coupled changes to the metamodels were metamodel-independent [71]. Supporting reusable and metamodel-independent adaptations with corresponding migration is therefore important. A way to support this is to allow the directives and operations to be generic, i.e. by using variables for the elements affected by the directives or operations. A template instantiation or adaptation strategy can then be applied to any metamodel for which there exist mappings between every variable of every directive or operation and metamodel elements of the correct type (as dictated by the semantics of the directives or operations). This in turns means that the same (derived) migrations can be applied on models of all metamodels that can be adapted by the generic template instantiation or adaptation strategy. The generic variants of the directives and operations would be based on already defined directives and operations. Hence, the generic directives and operations do not induce new challenges regarding preservation of type-safety or model conformance. For Artefact

A, supporting generic directives requires that the template instantiation directives can (optionally) be specified in a separate document and not only in the templates and packages. Instantiating a template thus translates to selecting a template and associating it with the (generic) directives. The design of the metamodel template mechanism supports both internally and externally defined directives (the prototype only supports the former). A template instantiation or adaptation strategy comprising generic directives/operations would identify a structural pattern(s), i.e. metamodels sharing this pattern(s) would be candidates for adaptation (including corresponding model migration). This means that it may be possible to infer some of the mappings between the variables and the metamodel elements. For Artefact C the adaptation strategy is separately specified. In Research Report I we discuss an approach for applying the refactoring operations of Artefact C in a generic manner by searching a metamodel for compatible structure. Defining generic directives and operations by means of metamodel types may also be a viable solution.

We have already argued for the limited number of instantiation directives and operations supported. All directives and operations, with the exception of code blocks in Artefact A, are declarative, i.e. the user only specifies *what* to be achieved, not *how*. This hides complexity and makes it easier to apply adaptations successfully. Providing means for specifying adaptations imperatively would support rare (and complex) scenarios (including corresponding migration). With respect to preserving type-safety this would typically be problematic. However, for some types of adaptation, it would be possible to specify migrations that allow re-establishing conformance automatically. As argued, we aim for supporting the most common forms of adaptations and corresponding migrations. For rare adaptations, it is always possible to utilise a GPL.

For the other artefacts reuse is not a primary concern. For Artefact D, reusing mappings between metamodels is typically not possible. For Artefact E, there is no other aspects regarding reuse than what is natively available by using the subtyping mechanism, e.g. that structure and code can be inherited and that input parameters of tools and artefacts can be typed to utilise polymorphism.

7.1.4 Separation of Concerns

A DSL comprises both syntax and semantics (behavioural and static). It is argued that separating the definitions of these concerns is advantageous since it allows different stakeholders to focus on each concern and it supports different variation points [56]. In addition, it is desirable to define behaviour independently of specific metamodels [42]. Artefact A supports separating these concerns. One template may define the syntax (classes and relationships), whereas another template may contain classes with operations defining the behaviour. The classes of the different templates can then be merged during template instantiations. The syntax and behaviour may evolve independently in distinct branches and be maintained by different stakeholders. This supports having metamodels with different semantics (semantic variation point).

For Artefact E, a similar separation of concerns can be achieved by inheriting pre-defined behaviour from supertypes, or alternatively by defining tools and runtime environments (e.g. an interpreter) based on the structure of a supertype that several metamodel types specialise. For the latter case, the metamodel of a metamodel type can then be purely structural whereas the operational semantics for executing models of this metamodel is defined in an interpreter. This way, the interpreter can define the operational semantics for several metamodels. In both cases, the behaviour can be "attached" selectively to the classes of a metamodel, i.e. the metamodel classes subtype/specialise classes of the supertype. The metamodel classes either inherit the behaviour directly (case 1) or the metamodel classes inherit the required structure to be compatible with an interpreter that contains the behaviour (case 2). The separation of behaviour from metamodels is similarly done in [42], and results in an additional level of indirection between structure and behaviour. A drawback of using subtyping to achieve this, as argued by the authors in [42], is that inheriting the behaviour from a supertype results in an intrusive change to a metamodel, which in turn requires the supertype to already exist. Also, the authors claim that this approach may not be feasible if several semantics for the metamodel are needed. We have already illustrated how the latter limitation can be addressed by using multiple inheritance (refer Chapter 6). The severity of the limitation regarding intrusively changing a metamodel depends on how metamodel types are used. In our case, we want to utilise (and reuse) pre-defined tools or behaviour that are defined relatively to a type that metamodel types subtype; the tools or behaviour already exists, and is used in the metamodel definition phase. Therefore, in this case, we argue that the intrusive change of a metamodel is acceptable.

7.1.5 Integration of Heterogeneous Metamodels and Models

All artefacts are designed to work on metamodels and models that are created using the same metamodelling framework, i.e. Kermeta and EMF. There are incentives for supporting the use of different metamodelling frameworks, or technological spaces, in unison. In particular, different DSLs may be expressed using different technologies. Also, it is argued that DSLs should be kept separately in their own space, where their models are connected using dynamic links [101].

In Related Paper II, we have proposed a way of integrating the operational semantics of metamodels by considering metamodels as services. The approach works by defining service contracts, i.e. provider and consumer interfaces. An interface relates to one or more class operations in a metamodel. The underlying idea is that the operational semantics of the various metamodels are integrated in a service-oriented manner as dictated by the interfaces, i.e. a provider interface has to be matched by a consumer interface. A metamodel service may have an arbitrary number of both provider and consumer interfaces which would allow integrating the operational semantics in different ways. Service architectures can comprise an arbitrary number of metamodels, i.e. the operational semantics of a metamodel may be integrated with the operational semantics of several other metamodels. The approach is particularly useful for decomposing the modelling of a system into several concerns (viewpoints).

The approach utilises proxy model objects which give a transparent way of invoking operations. A proxy object represents one or more objects in another model, and is inferred by the information in a provider interface. This means that all the operations of the provider interface can be invoked on the proxy object. Operation invocations on the proxy object (in a model MA of metamodel MMA) are translated to XML-based messages which is then resolved as invocations of the actual operations on one or more objects in model MB of metamodel MMB. The consumer interface specifies the operations that must be available on objects that reference the proxy object, i.e. for the purpose of callback. By using a proxy object, it appears that the models are composed through the reference(s) to the proxy object (in model MA). The specific models of the metamodels that should be integrated are specified by the user. The actual (dynamic) link between two models is only maintained during runtime. A proxy object is an instance of a temporarily added metaclass (added to MMA).

The service-oriented nature of the approach allows different concerns to be modelled without entanglement of concepts. A model describing one concern may utilise services of one or more other models by invoking operations on proxy objects. Since XML-messages are used for transferring operation invocations it is possible to support heterogeneous model integration as long as all the metamodelling frameworks allow defining service contracts and a way of resolving XML-messages to operation invocations and vice versa.

Both the framework for integration of operational semantics and the service-oriented approach rely on a conceptual integration of concepts. Conceptual integration may simplify integration since it is not necessary to consider in depth how the concepts are implemented [43] (yet knowing their semantics is essential).

7.1.6 Deriving Views and View Synchronisation

We have already discussed separation of concerns and heterogeneity. The main challenge in this respect is being able to support modelling according to different views (languages), which becomes important as the complexity of a software system increases [106]. We have discussed multi-view modelling with respect to (meta)model composition. A disadvantage when composing models describing different views is the inability to synchronise these with the resulting model from the composition [106]. This means that changes performed on the resulting model can not be propagated to the source models, or the other way around. We have not addressed this concern for Artefact A or Artefact C.

An alternative take on multi-view modelling is to derive views based on existing metamodels and models, e.g. as discussed in [101] and [106]. The artefacts discussed in this thesis were designed based on premises (stipulated as requirements) related to type-safety and correctness, support for evolution (including model migration) and reuse. In other words, providing means for deriving new views has not been a focus point. However, we will review the capabilities of the artefacts that support deriving new views.

Artefact D supports integrating models at runtime in terms of setting and getting attribute values and invoking operations on the cross of metamodel boundaries. This means that a new metamodel for a particular view can be constructed, where several (or even all) of the classes are proxies for classes in other existing metamodels. Since updating the state of proxy class objects is reflected in models of the existing metamodels (according to the mappings), we have a uni-directional form of synchronisation. Importantly, the new view can be constructed non-intrusively, i.e. without changing the existing metamodels. The use of proxy classes means that virtual models can be used. A virtual model contains proxies representing elements of other models [106]. That is, a virtual model does not contain concrete (domain) data; the data is contained in the model objects represented by the proxies. For Artefact D, virtual models can be realised if two conditions are fulfilled: 1) all classes of the view metamodel are proxies and 2) all setter and getter methods for the class attributes are overridden (in Java) to propagate/acquire values from the target objects (that the proxy objects represent). The objects of the proxy classes are used in the linking model as before to indicate what model objects should be updated when the virtual model is updated. It is possible to extend the artefact to update several concrete model objects based on "updating" one single proxy object. This may be interesting when the models cover overlapping views of some degree. An object of a proxy class is always "virtual" by default, which means that a model can be partially virtual, i.e. where some values are propagated/acquired to/from underlying models, whereas other changes are stored directly in the model of the view metamodel.

Similarly, the approach discussed in Related Paper II allows building a new metamodel for a particular view and connecting its operational semantics with the operational semantics of other metamodels in a service-oriented manner. On the contrary to Artefact D, it is only possible to invoke operations and not setting and getting attribute values. Artefact E also supports defining new views by utilising multiple inheritance, as discussed briefly. That is, a new view (metamodel type) comprising concepts from several metamodel types can be created by inheriting classes from the metamodel types. Model/view synchronisation is not available when using inheritance to define a new view.

7.1.7 Additional Artefacts

We have developed two additional artefacts which are discussed in research reports. The first artefact, as described in Research Report I, is an extension of Artefact C, comprising eight operations for refactoring of metamodels and an experimental approach for automatically detecting refactoring possibilities in metamodels. The approach is based on instantiating the operations with structures from a metamodel and evaluating whether the operations can be applied successfully (based on the premises of the operations). The possible refactoring patterns are presented to the user who selects which patterns to apply. The refactoring of the metamodel is reflected on existing models which maintain conformance with the metamodel.

In Research Report II, we discuss a language for integration of metamodels' operational semantics. The approach works by exporting and importing metamodel structure; exported structure, e.g. attributes and operation signatures, of one metamodel are imported *within* class operations of another metamodel. The imported structure can be used to define statements in the operation. That is, the value of an attribute can be read or set from the operation, and an operation can be invoked. Since structure is imported in operations, the approach does not induce impacts on other artefacts since the relations between a metamodel and other artefacts do not consider the contents of operations. It is possible to attach contracts to the exported structure, e.g. stating that the value of an attribute has to be within a certain range. The language has been implemented as an executable prototype. We will refer to this artefact as Additional Artefact II.

7.1.8 Usage Scenarios for the Artefacts

Artefact A/B (seen as one artefact) and Artefact C support certain kinds of metamodel adaptations, i.e. constructive/additive and corrective/refactoring adaptations. In addition, the artefacts support different kinds of metamodel compositions and model migration/composition. The limited set of directives and operations means that the artefacts, in their current versions, are best-suited for a specific kind of usage scenarios, i.e. each tool has strengths and weaknesses [75].

Eight categories for choosing a model migration tool (with preceding adaptation) is given in [75]. These are *1) frequent, incremental evolution*, *2) reverse engineering*, *3) modelling technology diversity*, *4) quicker migration for larger models*, *5) minimal dependencies*, *6) minimal hand-written code*, *7) minimal guidance from user* and *8) support for metamodel-specific migration.* For Artefact A/B and Artefact C we believe the optimal usage scenarios are frequent, incremental (additive) evolution where there is minimal need for hand-written code and minimal guidance from the user, and where metamodel-specific migration is common, i.e. 1), 6), 7) and 8). In addition, the Artefact A/B are specifically designed for metamodels that also define operational semantics.

For Artefact D and Additional Artefact II the usage scenarios are for integrating the operational semantics of metamodels that define such, where it is important to minimise impacts on artefacts and minimise changes to the metamodels. The artefacts work well for prototyping integration between metamodels and in testing scenarios. Also, situations where metamodels should only be loosely coupled are ideal.

Metamodel types, i.e. Artefact E, are useful for creating language variations that can utilise a common set of tools and behaviour.

Artefact A/B and Artefact C support similar usage scenarios. The main differences are that Artefact A is type-safe, supports metamodels with operational semantics, and are based on the use of templates.

Artefact D and the Additional Arterfact II also support similar usage scenarios. However, the way integration of operational semantics is achieved indicates where the artefacts are best suited. For Artefact D, integration is achieved in terms of using proxy classes that override classes of other metamodels. For the Additional Artefact II, integration is achieved by exporting structure from one metamodel that is imported within class operations of another metamodel. Structure can be imported within arbitrary operations. It is also possible to import structure from several metamodels within a single operation. This means that the approach is highly flexible. However, it also makes the approach potentially more difficult to reason about and understand for all stakeholders of a project. Additional Artefact II does not induce any impacts on artefacts as long as the specification of what structure to be exported is specified in an external document, i.e. not within the metamodels' definitions (this is not supported by the prototype, but it is straightforward to implement). Artefact D may induce impacts if a proxy class is added, i.e. if none of the existing classes in a metamodel can be used as a proxy class. Whether to use Artefact D or the Additional Artefact II depends both on the flexibility required, what stakeholders that are involved and how critical it is to not induce impacts on other artefacts, e.g. for prototyping and testing scenarios impacts may be more acceptable.

7.1.9 Combining the Artefacts in the Same Modelling Environment

The artefacts yield different ways of composing and adapting metamodels. Each artefact operates according to a set of rules, i.e. along a dimension. By combining the artefacts we are able to compose and adapt metamodels along several dimensions. For instance, metamodel templates may contain a combination of metamodels and metamodel types. A template and the outer class of a metamodel type may utilise generic type parameters. Moreover, metamodels produced by instantiating templates may be integrated by creating mappings between them. Also, by extending a metalanguage with constructs for exporting and importing structure and for treating metamodels as services we can integrate metamodels that are created in different metamodelling frameworks.

Furthermore, model types [53] and the built-in aspect weaving mechanism of Kermeta may be used in combination with the artefacts. Similarly as for metamodel types, model types may be used when specifying generic template directives. Specifically, a generic directive may list one (or more) model types that specify the structure that has to be matched in a metamodel. This includes listing several different model types, where each of the model types gives an optional set of required structure/classes that needs to be matched.

We have mentioned how a revised version of the metamodel template mechanism may have generic type parameters. A metamodel type can also be used to specify a type parameter, i.e. the valid arguments have to be subtypes of the metamodel type. Alternatively, model types can be used in a similar way. By using model types the arguments do not have to be derivatives of a metamodel type. Instead, the arguments/classes may be chosen as long as they can be matched successfully with respect to the model type used as type parameter. The result of instantiating a metamodel template with an argument(s) is a customised metamodel including the provided arguments in its definition. In particular, this brings interesting options to the table with regard to customising/integrating operational semantics.

Another possible direction is to use a metamodel type to define the operational semantics of a corresponding model type. That is, the classes of the metamodel type reflect those of the model type and vice versa. This allows providing operational semantics for model types.

7.2 Evaluation

In this section we evaluate the artefacts based on metrics proposed by others in the field and according to the requirements stipulated in Chapter 5. We also give our thoughts on usability of the artefacts and how easy the artefacts are to learn.

7.2.1 Evaluation with Respect to Metrics

Here we will evaluate the artefacts with respect to metrics proposed in the literature. We are not aware of specific metrics for evaluating Artefact D, Artefact E and the Additional Artefact II.

Composition

It is not easy to compare composition approaches [20]. It may therefore be better to evaluate an approach based on requirements and desirable properties. The authors of [32] have identified four requirements for model composition frameworks, which are: *the ability to specify corresponding elements of different models*, *means for describing how the corresponding elements are merged and composed*, *means for describing how* *non-corresponding elements are added to the target model* and *ways to manage and reuse correspondences and merge/composition operations*.

Artefact A and Artefact C meet three of the requirements. The forth requirement is not met. The correspondences between metamodels are specified manually in the instantiation directives and operations. As an example, when merging two classes the correspondence is established by referring to the two classes in the merge directive/operation. Since correspondences are not specified separately, we do not see that reuse of such makes much sense for Artefact A or Artefact C. We have previously discussed reuse in terms of generic directives and operations.

The authors identify two additional desirable requirements, i.e. a *composition framework should provide ways of minimising the effort needed to specify composition operations*, and a *composition framework should not be dependent on a specific metamodel for the purpose of supporting backwards compatibility, extensions and a variety of tools*. None of the artefacts consider these requirements explicitly.

Finally, the authors list four requirements for model composition tools: *the ability to verify composition operations*, *means for executing the operations*, *a debugger for analysing the composition* and a *mechanism for serialising models*. The prototype for Artefact A satisfies the first, second and fourth of these requirements.

A framework for evaluating aspect-oriented modelling approaches is proposed in [92]. The framework comprises a catalogue of evaluation criteria. We will use a selection of these criteria to classify the artefacts, i.e. we focus on criteria pertinent to concern composition. This allows comparing the artefacts with the approaches evaluated in [92]. We will later compare the artefacts to one of these approaches. The framework is primarily designed to evaluate UML-based approaches. Therefore, there are some criteria that are not applicable to the artefacts. The criteria are primarily relevant for Artefact A and Artefact C. However, we will relate to the other artefacts where appropriate.

Composition Mechanism The composition mechanism used by Artefact A and Artefact C is known as a compositor (CMP), i.e. classes are merged. In addition, the artefacts support composing metamodels by defining relationships between classes.

Element Symmetry The artefacts support concerns that are decomposed symmetrically. This means that all concerns, both non-crosscutting and crosscutting, are co-equal first-class elements that have the same underlying structure (as opposed to having base and aspect concerns of different structures). We have discussed how asymmetric decomposition of concerns can be supported as an extension of Artefact A.

Rule Symmetry Both Artefact A and Artefact C specify directives/operations for composition in a symmetrical manner. This means that directives/operations are specified outside of the concerns (metamodels) that are composed. For Artefact A, it may seem that the (composition) directives are specified asymmetrically since they are specified within templates. However, the directives are not part of the concerns and could have been specified in a separate document. This would not have changed the composition process.

Composition Symmetry Since we have element symmetry, we also have composition symmetry. This means that concerns, i.e. metamodels, are composed symmetrically. There are no constraints (e.g. following a base/aspect scheme) with respect to how concerns can be composed.

Composition Artefact A and Artefact C compose concerns statically. Artefact D and Additional Artefact II compose concerns in both a static and dynamic manner: the mappings between metamodel structures are created statically, but the operational semantics of the metamodels are first integrated (according to the mappings) when the models are executed.

Conflict Resolution Artefact A relies on conflict avoidance, which means that conflicts need to be detected and resolved manually before they occur. This applies for Artefact C as well. However, Artefact C additionally detects some naming conflicts which are resolved automatically.

Structural Composable Element Classes are the only elements that can be merged by Artefact A and Artefact C.

Behavioural Composable Element No elements describing behaviour can be composed (the artefacts operate on class models, which are structural models).

Match Method There is no support for automatic matching of elements. A match between two elements is specified manually. The original syntax of Artefact A uses *match-by-name* to initiate a class merge. Artefact D is based on *match-by-signature*, as the mappings created between elements can be seen as matching of their signatures - which have to be similar/identical.

Merge For Artefact A and Artefact C, a merge of two classes is specified manually using a merge directive or merge operation, respectively. Artefact A supports merging several classes using one merge directive, whereas Artefact C can only merge two classes with each use of the merge operation. Merging of multiple classes have to be addressed by using several merge operations.

Override Artefact C allows overriding classes. This means that a class may replace another as long as this new class contains at least all the properties and operations of the class it replaces. Artefact D allows overriding classes using proxies.

Bind Binding between composable elements (i.e. in a template parameter manner) is not supported.

Abstraction The composition directives/operations have a low level of abstraction. In other words, the specific classes that are intended to be merged are specified (in contrast to a high level where e.g. composition is specified only by identifying two packages whose contents should be composed).

We will compare the artefacts with the approach of [25], which is one of the approaches being evaluated in [92] that most closely resembles the artefacts. Specifically, we will relate to how the approach supports composition of class diagrams. We will base the comparison on the evaluation of the approach in [92].

Similarly as for Artefact A and Artefact C, the approach of [25] composes class diagrams using the compositor mechanism. The approach differentiates between base and aspect models. An aspect model is a parameterised package that contains class diagram templates, communication diagram templates and sequence diagram templates. A context-specific aspect is made by binding the elements of the templates to application-specific elements. The approach has element, rule and composition symmetry, and allows matching elements automatically based on their signatures. This implies that structural conflicts are identified as well. An aspect model is statically composed with a base model. Conflicts are addressed by using model composition directives and element directives. The former allows specifying the order of how aspects are composed with the base model. The latter allows adding, removing and replacing model elements. The composable elements are Classifier, Association, Operation and Model. Elements can be composed using merging or overriding. The approach supports composition at both a high and low abstraction level. For class diagrams, composition is performed at a low level.

Artefact A is also based on using templates. The difference compared to the templates used by [25] is that metamodel templates do not have parameters in the traditional sense. One way to use parameters for metamodel templates is in the form of generic type parameters. This would allow customising a template for a specific application. In Paper I we discuss similarly how this can be achieved for nested classes. The approach of [25] allows describing the order of how aspects will be composed with the base model. Using Artefact A, this is not required since concerns/aspects can be composed directly with each other, i.e. we do not differentiate between base and aspect models. An interesting possible direction for metamodel templates is to allow defining other types of models/diagrams within the templates, e.g. sequence models. This would give an alternative for specifying behaviour, by means of how messages are sent between operations. It would require support for asymmetric composition in addition to the already supported symmetric composition mechanism. The approach of [25] supports merging of properties and operations. Supporting such for Artefact A and Artefact C would be possible by differentiating between two operation modes, as previously discussed.

Migration

Four requirements for automated coupled evolution are proposed in [71]. These are pertinent to: *reuse of migration knowledge*, *expressive/custom migrations*, *modularity of coupled changes* and *the availability of a history.*

Artefact A/B and Artefact C do not support reuse of migration knowledge with the purpose of supporting metamodel-independent changes. We have previously discussed a way of supporting directives and operations of these kinds by utilising variables for model elements in the directives/operations.

Artefact A/B and Artefact C do not support specifying custom migrations. The reasons for this are the same as those regarding adding additional directives/operations.

The requirement for modularity states that there should be support for specifying a coupled change (adaptation and migration) independently of other coupled changes so that the coupled changes do not affect each other. The artefacts fulfill this requirement.

The artefacts support the requirement regarding history, which says that it should be possible to migrate models separately from the metamodel adaptation, e.g. because the models may be distributed. This requires that information (a history) for migrating models is generated and made available to the user(s).

Nine criteria for comparing model migration approaches are given in [75], i.e. *construction*, *change*, *extensibility*, *reuse*, *conciseness*, *clarity*, *expressiveness*, *interoperability* and *performance.* The paper compares four approaches with respect to the criteria. We will use these criteria to evaluate the Artefact A/B and Artefact C in the light of this comparison. Specifically, we relate and compare the artefacts to the characteristics of COPE [63] which is one of the four approaches being evaluated. The reason we focus on COPE is that the other approaches are too different from the artefacts to yield interesting reference approaches. COPE is a recognised operator-based approach which resembles the artefacts. It is the basis for the Edapt framework[3]. (We base our comparison with COPE on the evaluation of this approach in [75].)

Constructing the Migration Strategy Artefact A/B and Artefact C derive a migration strategy based on the directives and operations that are applied on metamodels. This resembles how COPE uses a history of applied operations to generate the migration strategy. The evaluation of COPE states that it is not always straightforward to know how to sequence operations in order to get a correct migration. For Artefact A/B and Artefact C this does not seem to be problematic. The likely reason for this is that the artefacts only support a limited number of directives/operations, in particular, destructive directives/operations (and moving of elements) are not supported. The evaluation states that reverse engineering a large metamodel can

[3]http://www.eclipse.org/edapt

be challenging. For Artefact A, this is likely not to be problematic since the instantiation directives specify every change that has been applied to a metamodel, which collectively describe how the metamodel can be reverse engineered. All the instantiation directives are available by inspecting a template hierarchy. A similar reasoning applies for Artefact C. An adaptation strategy together with the generated effects (i.e. the name mappings) specify every change that has been applied to a metamodel. However, the adaptation strategy and effects need to be stored for later reference if reverse engineering should be supported.

Changing the Migration Strategy COPE supports fixing migrations that are incorrect. It is also possible to chain migration strategies which means that models of any previous metamodel version can be migrated to the current metamodel version. For Artefact A and Artefact C, the transformations for migration are derived based on the instantiation directives and operations that are used. Hence, migration only makes sense when the metamodels in question have been adapted (and composed) successfully. If this is the case, the migration will also be successful (only formally proven for Artefact C). Artefact A supports chaining of migration strategies as well. This means that models conforming to a previous metamodel version can be migrated to the current metamodel version (i.e. of the same branch). This can easily be achieved by specifying the source and target metamodel in a template hierarchy, which results in the generation of a chained migration strategy (basic transformations). The same applies to Artefact C. However, as stated previously, this requires the adaptation strategies to be appended after each other and the effects for all the adaptation steps (metamodel versions) to be collected and normalised (i.e. intermediate names for elements are discarded). Together the composite adaptation strategy and collected effects contain all the information necessary to migrate models of a previous metamodel version.

Extensibility The library of available operations in COPE is extensive. It is also possible for a user to write his/her own operations using a transformation language. For Artefact A/B and Artefact C we have a limited set of directives and operations, respectively. It is not possible for the user to write his/her own directives/operations. Artefact C supports rule inheritance.

Reuse COPE allows reusing coupled operations (i.e. capturing both metamodel evolution and model co-evolution). User-specified operations can also be reused. Artefact A/B and Artefact C support reuse of directives and operations. However, the specific elements referred to in the directives/operations need to be updated to match those of another metamodel(s). We have discussed how generic variants of the directives and operations can be made by using variables.

Conciseness For COPE, each operation application is specified using one line of code. The same applies to directives and operations for Artefact A and Artefact C. The only exception is the definition of code blocks for Artefact A which may have an arbitrary number of lines of code. An example of adaptating a Petri net metamodel (with migration) is used to measure the number of operations needed by each mechanism. COPE requires using 11 operations to perform the intended adaptation (with migration). Performing the same adaptation using Artefact A sums up to 10 instantiation directives if we exclude the default *instantiation* directive which is always present, and count four usages of a fictive directive for deleting references (since this is not supported). In addition, three new classes need to be created (added without using the metamodel template mechanism). For Artefact C we need to use 12 operations (including four usages of a fictive operation for deleting references).

An example of adapting (with migration) the GMF Graph metamodel from version 1.0 to version 2.0 requires 76 operations and 73 lines of additional code using COPE. Artefact A and Artefact C do not have directives/operations for deleting references. This action is required for adapting the GMF Graph metamodel. However, we have used Artefact A and Artefact C for adapting the GMF Graph metamodel from version 1.0 to 2.1 (extracts of these metamodels are found in [102]) with the exception of deleting references. For Artefact A the adaptation requires 11 directives (1x class renaming, 2x reference renaming, 4x code blocks, 4x add code blocks) and adding three new classes. For Artefact C this requires using 11 operations (3x add class, 4x add property, 1x rename class, 2x rename property, 1x add superclass). In both cases, the resulting metamodel is well-formed. However, three old references (one regular and one bi-directional) are present which clutter the metamodel definition. Deleting these would require using a delete reference directive/operation three times. In addition, Artefact C does not have the ability to change a class from being abstract to concrete. This is a trivial action, but would require using one more operation.

Note that the counts for the directives/operations for the two cases (Petri net and GMF) only reflect the adaptation of the metamodels and do not address specific concerns with respect to migration (as automatic migration when deleting elements poses certain difficulties). Therefore, comparing the numbers with those of COPE is not accurate.

Clarity A release history records what operations that have been applied when using COPE, i.e. it is clear what changes that have occurred. Similarly, a template hierarchy created using Artefact A contains all instantiation directives that have been used to produce a certain metamodel variant or version. By traversing the hierarchy it is possible to collect all directives that represent the differences from one metamodel version to another. Artefact C does not store what operations that have been used. To achieve the same level of documentation, it is required to manually save the adaptation strategies and generated effects.

Expressiveness COPE was able to adapt/migrate both test metamodels/models given (Petri net and GMF Graph). Adaptation and migration of the Petri net metamodel and models was achieved using only pre-defined operations. The GMF Graph metamodel/models were adapted/migrated using pre-defined operations and two custom migration actions. Artefact A and Artefact C do not have a directive/-operation for deleting references. They can therefore not adapt the specific Petri net metamodel as desirable. Adapting the metamodel using Artefact A and Artefact C gives a well-formed metamodel. However, there are additional (obsolete) references that clutter the metamodel definition. The artefacts are able to adapt the GMF Graph metamodel 1.0 to a well-formed 2.1 version. However, as with the Petri net case, we end up cluttering the metamodel with obsolete references. Also, the artefacts do not support specifying custom migrations using imperative constructs.

Interoperability COPE (the tool) has certain dependencies, e.g. it requires Eclipse and EMF. It does not require having both source and target metamodels for creating a migration strategy. The source metamodel may be reverse engineered from the target metamodel. Artefact A (i.e. the prototype realising the mechanism) is built on EMF and uses the *Textual Modeling Framework*[4] (metamodels and models have to be conformant with Kermeta/Ecore). However, the artefact can be realised using other kinds of platforms and frameworks. For generating the migration strategy and performing the migration, both the source and target metamodels (templates) need to be available. Artefact C is not implemented as an executable tool. The source and target metamodels need to be available for generating the migration strategy. The target metamodel is required when performing the model migration.

Performance We have not evaluated the performance (i.e. execution time) of the artefacts. This is not relevant since we only have prototype implementation of the artefacts which do not necessarily yield the fastest execution times possible.

Usability and Learnability

Usability and learnability can not be measured without empirical (user) studies. We have not focused on this. However, we believe that the artefacts are straightforward to use - specifically Artefact A/B, Artefact C and Artefact E. Artefact A/B and Artefact C use declarative directives/operations, whereas Artefact E is based on class nesting which is a familiar concept. The conciseness of template instantiations (Artefact A) and adaptation strategies (Artefact C) are also satisfactory, as previously discussed.

[4]http://www.eclipse.org/modeling/tmf

Involving non-technical stakeholders in the different phases of software development is considered important. For Artefact A/B, Artefact C and Artefact D we think this is possible without too much difficulties. This further depends on whether appropriate, simple to use (graphical) editors are created.

Validation of Artefacts on Industrial Cases

A limitation of the work is that we have not validated the artefacts on industrial (large-scale) cases. Case studies are important in order to learn how scalable and robust the approaches are, and for identifying limitations, e.g. with respect to missing directives/operations. Also, such evaluation makes it possible to estimate cost-effectiveness and return on investment which are important concerns regarding applicability of an approach. Obtaining industrial data is very difficult [16]. In particular, getting access to models of industrial metamodels is not easy. There exists a library of metamodels for experimental use[5]. However, there is just a few metamodels that are available in different versions.

7.2.2 Evaluation with Respect to Requirements

Here we will discuss to what degree the artefacts meet the requirements stipulated in Chapter 5. The requirements are not applicable for all the artefacts. It will be clear from the text which artefacts the requirements apply to.

Specific Requirements

1. *Metamodel composition mechanisms should automatically ensure and assert the correctness of their application to the extent possible.* Artefact A achieves this by providing instantiation directives that preserve type-safety. Type-safety holds regardless of how the directives are sequenced (assuming no conflicts occur). Artefact C addresses this by using operations that are designed to ensure a correct composition. The operations are formalised in rules that state the premises for achieving a correct composition. If the analysis of an adaptation strategy, with respect to a number of metamodels, is successful then correctness of the composition (and adaptation) is guaranteed.

2. *Composition of operational semantics should be supported for metamodels that define such in class operations.* This is achieved for Artefact A by allowing to override operations, which in turn allows composing the operational semantics of metamodels. Artefact D addresses this by forwarding operation invocations between the operational semantics of different metamodels. Additional Artefact II allows importing structures within operations, which means that the operational semantics of different metamodels can be composed, i.e. the imported structures can be used in statements of the importing operation.

[5]http://web.emn.fr/x-info/atlanmod/index.php?title=zoos

3. *It should be possible to statically type-check metamodel compositions for metamodels that specify operational semantics in class operations.* Artefact A addresses this by relying on instantiation directives that preserve type-safety. For Artefact D this is ensured by verifying that all the types of the unification points between the metamodels are compatible or equivalent. Additional Artefact II meets this requirement by checking that imported properties and operations have the correct types with respect to the statements where they are used. This includes checking that the actual arguments used when invoking an imported operation are compatible with the formal parameters of the operation. The artefacts rely on static type-preservation/-checking.

4. *A metamodel composition mechanism should support composing an arbitrary number of metamodels simultaneously.* This requirement is met by Artefact A, Artefact C, Artefact D and the Additional Artefact II. Artefact A and Artefact C support symmetric composition. The other two artefacts compose/integrate the operational semantics in an asymmetric manner, e.g. a metamodel that uses a proxy class can be seen as a base model. However, a kind of symmetry can be achieved by using a two-way integration of the operational semantics, i.e. both metamodels take a base and aspect role simultaneously.

5. *Metamodels should be organised in a way that eases reuse and avoids redundancy of metamodel variations.* For Artefact A, this is achieved by organising templates in hierarchies. The hierarchies encode the history of evolution in a tree-like structure. No other artefact addresses this requirement specifically.

6. *Approaches for composing metamodels should take artefacts that are dependent on the metamodels into consideration; in particular the models.* Artefact A/B and Artefact C address this when it comes to models by supporting (semi-)automatic model migration, i.e. model conformance is re-established automatically after metamodels are adapted and composed. Model composition is also supported based on the transformations and effects that are generated. This requires specifying what models (and objects) that should be composed. We have implemented a transformation engine for Artefact B that allows composing two models by merging their root node objects. Artefact D and Additional Artefact II fulfill this requirement since using these does not impact other artefacts significantly. This is achieved by defining mappings between metamodels externally (unification model) and by importing structures within operations. The effects generated by Artefact C (and the adaptation strategy) may be used for co-evolving other artefacts. We have not evaluated this possibility.

7. *Approaches for metamodel composition should support adapting metamodels as part of the composition operation.* Artefact A and Artefact C meet this requirement by providing instantiation directives and operations, respectively,

for adapting metamodels. Adaptation can be performed both before and after composition.

8. *Mechanisms supporting adaptation of metamodels should support both metamodel-specific and metamodel-independent adaptations (coupled changes).* Artefact A and Artefact C only support metamodel-specific (coupled) changes. We have discussed how metamodel-independent (coupled) changes can be supported by generic versions of the directives and operations.

9. *Migration approaches should produce conformant models automatically, to the extent possible, and verify correctness of the migration.* Artefact B partially meets this requirement, whereas Artefact C meets the requirement. Artefact B produces conformant models automatically by deriving transformations which when executed migrate existing models. However, we have not proven that the derived transformations always produce a correct migration (hence the partial fulfillment of the requirement). For Artefact C we have proven that migration will always yield conformant migrated models. That said, it may be required to manually revise generated default objects and values to ensure a sound model that properly reflects properties of the problem domain.

10. *It should be possible to type metamodels.* Artefact E allows metamodels to be typed, and thereby supports using object-oriented mechanisms like subtyping and polymorphism for metamodels.

General Requirements

1. *Approaches for model management should be easy to use and reduce complexity; not introduce new layers of complexity.* Artefact A provides a simple and intuitive syntax for the instantiation directives, which we believe are straightforward to use. Sequencing the directives is not challenging. The only requirements such wise is that renaming of properties and operations, with the purpose of avoiding conflicts, is performed before classes are merged, and code blocks have to be defined before they are added to a class. The complexity regarding preservation of type-safety, e.g. propagation of names, are handled behind the scenes and is therefore not a direct concern for the user. Using Artefact C implies specifying an adaptation strategy comprising declarative operations. We have not evaluated whether there is a difficulty associated with sequencing the operations for larger adaptation strategies. Artefact A/B and Artefact C migrate models (semi-)automatically. The only required action by the user is to specify what models that should be migrated or composed.

2. *Management of models should be performed using declarative languages to the extent possible.* Artefact A and Artefact C use declarative directives and operations, respectively. Artefact A also allows creating code blocks with imperative code. For Artefact D, specifying mappings between the structures of

metamodels can also be seen as a form of declarative language. However, the model code associated with a metamodel utilising proxies needs to be updated; which is achieved using imperative constructs of Java.

3. *Approaches for model management should be applicable to as many (meta)-models as possible.* The artefacts support metamodels of arbitrary problem domains. The directives and operations of Artefact A and Artefact C, respectively, support metamodel-specific adaptations and compositions. Artefact D supports specifying mappings between arbitrary types of metamodels.

4. *Approaches for model management should be supported by development tools and editors.* We have not focused on defining tools or editors for the approaches. In particular, providing editors for specification of the adaptation strategy (Artefact C), and the M1-/M2-mappings (Artefact D) would improve readability and ease of use. That said, the prototypes of Artefact A and the Additional Artefact II provide textual editors which improve readability.

7.3 Related Work

The papers and research reports discuss related work pertinent to the various artefacts, and try to differentiate the artefacts from similar approaches. In this section we discuss recent work that has not been discussed in the papers or research reports. This section can also be seen as an extension of the state of the art in Chapter 3.

In [101], the authors discuss a way of integrating models in a similar manner as supported by Artefact D and Additional Artefact II, and discussed in Related Paper II. The approach works by creating a virtual model in a conceptual space by reusing features from models in technological spaces. A virtual model is made of local concepts (instances of a generic modelling language) that are connected to the concepts of models in the technological spaces. Additionally it is possible to define new concepts in the virtual model that do not belong to any of the technological spaces. The features in the models of the technological spaces are accessed using technological connectors (representing different modelling paradigms), i.e. it is possible to read and write (and thus synchronise) information between models in the technological spaces and the model in the conceptual space. Reuse of features from the technological spaces can be achieved either by creating a dynamic link between a concept in the virtual model and a concept in a technological space (which ensures synchronisation), or by using a proxy for a concept in a technological space. There is a bi-directional connection between concepts in models of the technological spaces and concepts in the virtual model, which means that updating either model is reflected in the connected model. The approach supports models defined using heterogeneous formalisms and different paradigms. The virtual model cross-cuts the models of the technological spaces, i.e. it represents a combined model of a multi-model system. Artefact D allows connecting metamodels and models using mappings. This in turn supports setting and getting values for attributes, and invoking operations. Similarly to what is discussed in [101], it is possible to construct what is referred to as a virtual model (by first creating a new metamodel) and map the concepts of this model to concepts of models conforming to existing metamodels. The main difference is how the approach of [101] supports heterogeneous models (i.e. different paradigms) and connecting concepts bi-directionally. Also, the approach focuses on integration of structural models, whereas Artefact D focuses on integration of metamodels and models that have an associated operational semantics. That is, the integration is achieved at runtime. In Related Paper II we discuss how services can be used to support integration of heterogeneous models (for paradigms that support operations).

EMF Views [106] is an approach for deriving modelling views non-intrusively based on existing models (and metamodels). The views are non-materialised, which means that actions performed on the view elements are forwarded to the underlying models (only changes to attributes are propagated to the models). A *viewpoint* is a description of concepts for a specific perspective that are collected from one or more metamodels, potentially elaborated with new interrelations. A *view* is a collection of virtual elements (proxies) connected to actual elements in the models of the metamodels. A viewpoint is specified using a query language (or eventually manually) that is oriented around three operations: *project*, *select* and *join*. Projection is used to specify what classes/features from the metamodels that should be included in a view(point). Selection allows specifying conditions that need to be satisfied by model elements for these elements to be included in the view. The join operation specifies how the elements from the different models should be linked. The links are stored in a weaving model. As discussed, Artefact D allows building a new (viewpoint) metamodel whose concepts can be mapped to those of existing metamodels. The linking model specifies what objects (whose Java runtime object equivalents) that are to be connected during runtime. This means that a new view can be constructed. However, the main difference is that integration between models only happens at runtime. Also, synchronisation is only partially available. That is, only changes to attributes in the new (view) model is propagated to the models of the existing metamodels and not the other way around.

The authors of [108] discuss a lightweight mechanism for specifying extensions to metamodels. The mechanism builds on EMF Views and allows defining extensions using a generic textual DSL comprising a pre-defined set of atomic extension operators (of type *add*, *modify* and *filter*). Metamodel extensions are realised in a virtual manner, i.e. the original metamodel is not changed intrusively. Instead, the extended metamodel utilises proxies that refer to elements in the original metamodel. Similarly, models of the extended metamodel is virtualised in terms of proxies which means that the concrete model data is propagated to existing models of the original metamodel. Since the DSL is defined in a generic way (with respect to formalism) it is possible for different modelling environments to reuse metamodel extensions.

Eclipse EMF Facet[6] is an approach for extending metamodels and models (by adding new features) virtually (e.g. when a model is open) in a non-intrusive manner. It is also possible to customise existing features. Query abstraction is supported which means that facets can be specified and customisations can be performed using query languages.

A consequence of using difference-based model migration approaches is that there may be generated several alternative migration strategies that support transforming existing models to valid instances of an evolved metamodel. This is also referred to as the graph isomorphism problem, i.e. different transformations are equivalent [112]. In [111], the authors present an approach that helps the user selecting the optimal migration strategy. The approach works by manually defining a weaving model between a model and classes in the evolved metamodel, i.e. the elements in the model that need to be migrated are linked to the respective classes in the evolved metamodel. The weaving model is then used as input to a transformation that outputs a feature model. The feature model shows variability in terms of differences between the alternative migration strategies, and shows potential conflicts between the strategies. Traceability between metamodel changes and corresponding migration alternatives is also supported. Artefact A/B and Artefact C are *prescriptive* approaches, on the contrary to *inductive* approaches [111], i.e. difference-based approaches. For prescriptive approaches models are migrated based on predefined operations, which avoids the difficulties of selecting a migration strategy after a metamodel has evolved. The authors state that the process of model migration should also consider aspects such as information erosion and finding a migration strategy that applies the minimal number of model changes, and not only aiming at preserving/re-establishing model conformance. For Artefact A/B and Artefact C these aspects are not of particular interest as long as directives/operations for deleting and moving elements are not used.

The authors of [112] present an approach for addressing co-evolution of models in terms of an automated multi-objective optimisation process, i.e. co-evolution is treated as an approximation based on heuristics. The process searches for a combination of model edit operations dictated by three objectives: minimising 1) the number of constraints that an evolved model violates with respect to the evolved metamodel, 2) the number of applied model changes and 3) the dissimilarity between the existing and evolved model. The model edit operations are of type *create/delete*, *retype* (an element is replaced by an equivalent element with a different type), *merge*, *split* and *move*. The best co-evolved models (based on an experiment) had a precision and recall greater than 86% and a manual precision greater than 92%.

An approach for co-evolution of models and transformations by establishing bi-directional transformations is discussed in [114]. The approach works by mapping two uni-directional transformations (e.g. as defined in ATL or ETL) to a

[6]http://projects.eclipse.org/projects/modeling.emft.emf-facet (version 1.0.0 was available in 2015)

bi-directional graph transformation model using a higher-order transformation. The higher-order transformation acts as a binding between the concepts of the uni-directional transformations and the graph transformation. This way, changes applied to one artefact can be propagated to the other and vice versa. Changes can be propagated between models, and between a model and a transformation (represented as a model). The models and/or transformation can reside on the same or different abstraction levels. Transformation of multiple dependent models is also possible. The approach of [114] resembles Artefact D to some degree, i.e. Artefact D supports a uni-directional synchronisation of models at runtime. Supporting bi-directional synchronisation should be possible as well, but we have not explored this possibility.

We have not discussed related work on composition approaches for languages defined using a grammar-based abstract syntax (as opposed to metamodels). The reason for this is that the challenges pertinent to languages written in the two types of abstract syntaxes differ. However, the approach discussed in [115] resembles our work to some degree and is therefore of enough relevance to discuss. The approach supports two types of language composition: *language aggregation* and *language embedding*. Language aggregation works by relating concepts of different languages with the purpose of interpreting models of the languages together yet keeping their definition, i.e. the abstract syntax trees (ASTs) independent of each other. Language embedding translates to composition of languages by embedding concepts of one of the languages into the other language according to declared extension points. The approach also supports *language inheritance* (can be seen as a form of composition in this case). Language inheritance is used for extending or refining a language where a new language can be defined by reusing and modifying concepts of an existing language. Language aggregation, and particular cases of language embedding, rely on using a *symbol table infrastructure*. The infrastructure allows acquiring information from referenced models and interpreting elements of one language as elements of another, e.g. when an element in one language references an element in another language by name (language aggregation). The infrastructure supports translating a concept of one language to a concept of another language. The approach is based on the MontiCore Language Workbench[7]. Language aggregation and language embedding resemble Artefact D where structure from one metamodel can indirectly be referred to in another metamodel via a proxy. A proxy also acts as an embedded concept from another language. Language aggregation and embedding also resemble Additional Artefact II. Additional Artefact II supports defining operations using imported structure and thereby allows integrating languages yet preserving their autonomous definition. And finally, language inheritance resembles metamodel typing, where a new metamodel/language can be defined in terms of concepts of existing metamodels. The main difference between the approach in [115] and the artefacts discussed in this thesis is that integration of models (with respect

[7]http://www.monticore.de

to language aggregation and embedding) can be achieved without impacting their validity (i.e. not breaking model conformance). Also, the artefacts are practically non-intrusive with respect to changing the metamodels, which essentially avoids impacting other artefacts like tools. Both Artefact D and Additional Artefact II use a form of identifier lookup, resembling the symbol table infrastructure of [115].

In [104][105] the authors discuss how graphical concrete syntaxes can be defined in terms of metamodel specialisation (also referred to as an extension) instead of metamodel instantiation. In the latter case a graphical syntax definition comprises a collection of objects that are instances of a type model. On the other hand, when using metamodel specialisation a specific graphical syntax definition is made by specialising the classes of a base metamodel which contains concepts for defining graphical syntax. The argument for using this approach is that the definition is direct (at the same metalayer), precise and more understandable than using metamodel instantiation. The authors argue that the innovation relies on the specialisation of a *whole* metamodel. The work relates to that of Artefact E, i.e. the theory for realising metamodel types. Similarly, we argue that a metamodel can be typed and specialised as a whole. The authors of [104][105] base their approach on UML, which means that associations can be redefined, i.e. the name of an association end can be changed and the multiplicity may be narrowed. For Artefact E we have not discussed this possibility; if the class operations of a metamodel contain code (or similarly if an interpreter is used) then redefinition of associations requires redefinition of operations as well. Otherwise, existing code will no longer execute correctly. For the approach discussed in [104][105] it is not possible to add new attributes or associations, but it is possible to specify OCL constraints. Other specialisation restrictions are also enforced, which the authors argue are required to ensure that the specialised classes preserve the intended meaning (though restricted) of the classes in the base metamodel. Relating to Artefact E, new contents can be added arbitrarily to the specialised classes. We argue that this is required for reflecting how concepts evolve. Also, it means that more metamodel adaptations are possible. We have not discussed OCL, but we foresee that it may provide a means of increasing the preciseness of specialised classes.

Chapter 8

Conclusions and Future Work

In this thesis we have studied metamodel composition and adaptation, integration of metamodels' operational semantics, (semi-)automatic model migration and metamodel typing. The study aimed to find flexible solutions to how metamodels can be composed and adapted (type-safely) with corresponding automatic model migration and composition, how impacts on artefacts in the metamodelling ecosystem can be minimised (as a consequence of metamodel composition and adaptation) and how metamodels can be typed to support reuse and variance. We here summarise the main achievements in terms of contributions and identify improvements for the artefacts.

8.1 Achievements

Based on the definition of package templates, we have derived a template-based mechanism for type-safe composition and adaptation of metamodels. All of the supported template instantiation directives preserve type-safety, which means that the metamodels may contain operational semantics which can be composed and adapted as well. As a continuation of this mechanism, we have created a framework that (semi-)automatically migrate or compose existing models, whose metamodel(s) have been adapted and/or composed, based on the instantiation directives that have been used. To the best of our knowledge, there are no other approaches for metamodel composition and adaptation that preserve type-safety to the degree our mechanism does. The metamodel template mechanism addresses many of the challenges regarding metamodel composition, e.g. it ensures the correctness of compositions, it allows composing an arbitrary number of metamodels simultaneously, it provides means for easy reuse where metamodel versions are organised in hierarchies and its use is oriented around easy to use directives.

In a similar approach, we formalise a framework for analysis of metamodel adaptation and composition with corresponding model migration and composition. We are not aware of any work that formalises these operations and proves that model conformance is re-established automatically by means of a framework for analysis.

We have created a framework for integrating the operational semantics of metamodels which can be used practically non-intrusively. This in turn results in minimal impacts on artefacts like tools and editors. The integration of the operational semantics is achieved by creating structural mappings between elements of different metamodels. The mappings allow the models/code to get and set values of attributes and invoke operations at runtime on cross of metamodel boundaries. Similarly, we have discussed an additional artefact comprising a language for exporting and importing metamodel structure, which also allows integrating the operational semantics of metamodels while creating few impacts on other artefacts.

A theory for defining metamodel types has been proposed. By defining a metamodel within a class, we are able to define a reference type that tools and other artefacts can be built around. A metamodel type that subtypes the reference type is compatible with the tools and artefacts (i.e. with respect to the concern of the supertype). This ensures that operational semantics and other forms for processors can be defined once and reused by several metamodels. It is possible to consider (and process) a metamodel type according to several views. Metamodel types support inheritance and specialisation of metamodels.

8.2 Future Work

We have discussed some limitations with the artefacts. This section summarises directions for future work according to these limitations. Specifically, there are extensions we believe would improve the value and applicability of the artefacts.

8.2.1 Extensions

Artefact A and Artefact C do not support deleting or moving structural elements in metamodels or changing multiplicities for class properties (Artefact C supports moving properties and operations in a class hierarchy under certain conditions). Extending the mechanisms with constructs for these actions would support a broader range of adaptation scenarios. However, these actions should be differentiated as an additional set of directives/operations since type-safety and automatic re-establishment of model conformance can not be guaranteed. A directive/operation that will not induce difficulties with respect to type-safety or automatic model migration is that of changing the lower bound of a multiplicity from zero to any number less or equal to the upper bound.

Having means for coding custom migration using imperative constructs is advantageous. Such constructs would complement the additional set of directives/-operations, e.g. by allowing describing how moved attributes or references can be populated with values and objects, respectively. As an example, moving an attribute can be seen as deleting one attribute and adding a new one (with the same name, type and multiplicity) to another class. The value(s) of the old attribute should be moved respectively in the model, i.e. to the object of the class to which the attribute has been moved[1]. This can be described as a mapping between the deleted attribute and the new attribute.

Supporting generic versions of the directives/operations of Artefact A and Artefact C is desirable. This would allow specifying adaptation/migration strategies that can be reused. Moreover, we have discussed a way of supporting asymmetric composition based on Artefact A which would allow specifying pointcuts with the purpose of weaving in crosscutting concerns.

Another extension that would be advantageous is to support specifying the specific default values for primitively typed attributes. This is supported by EMF/E-MOF, but not in Kermeta. For Artefact A and Artefact C specific default values can be supported by extending the grammar for attributes/properties. For Artefact C we would also require that the operation for adding properties is updated to allow specifying specific default values. This is not required for Artefact A as new properties are added in code blocks which are written in Kermeta.

EMF is one of the most popular metamodelling frameworks. Implementing a tool for supporting metamodel templates in EMF is therefore of interest. EMF is principally close to Kermeta. However, the metamodel structure and operational semantics (model code) are expressed in two different modelling spaces. This means that at tool has to reflect this separation as well.

Finally, providing graphical tools will further improve usability and readability.

8.2.2 Validation

Validating the artefacts on industrial use cases is required to solidify the artefacts and evaluate how applicable and usable the artefacts are. This requires Artefact C to be implemented as an executable prototype (we have only evaluated it analytically in terms of a proof, i.e. that model conformance can be guaranteed after migration). We have not formally proved that Artefact B always ensures that model conformance is maintained. A formalisation of this artefact would also make it easier for others to generalise from our approach.

[1]This is not necessary when an attribute is pulled upwards in a class hierarchy.

Bibliography

[1] I. Solheim and K. Stølen. Technology Research Explained. Technical Report A313, SINTEF 2007.

[2] T. Clark, P. Sammut and J. Willans. Applied Metamodelling: A Foundation for Language Driven Development (Third Edition). 2015. http://arxiv.org/abs/1505.00149

[3] D. Di Ruscio, L. Iovino and A. Pierantonio. Evolutionary Togetherness: How to Manage Coupled Evolution in Metamodeling Ecosystems. In Graph Transformations, LNCS volume 7562, pages 20-37. Springer 2012.

[4] B. Henderson-Sellers. On the Mathematics on Modelling, Metamodelling, Ontologies and Modelling Languages. In Springer Briefs in Computer Science. Springer 2012.

[5] C. Atkinson and T. Kühne. Meta-level Independent Modelling. In Proceedings of the International Workshop on Model Engineering (in conjunction with ECOOP). 2000.

[6] C. Atkinson and T. Kühne. The Essence of Multilevel Metamodeling. In The Unified Modeling Language. Modeling Languages, Concepts and Tools. LNCS volume 2185, pages 19-33. Springer 2001.

[7] J. de Lara and E. Guerra. Deep Meta-Modelling with MetaDepth. In Objects, Models, Components and Patterns. LNCS volume 6141, pages 1-20. Springer 2010.

[8] S. Kent. Model Driven Engineering. In Integrated Formal Methods, LNCS volume 2335, pages 286-298. Springer 2002.

[9] D. C. Schmidt. Model-Driven Engineering. In Computer volume 39, issue 2, pages 25-31. IEEE 2006.

[10] C. Atkinson and T. Kühne. Model-Driven Development: A Metamodeling Foundation. In Software, volume 20, issue 5, pages 36-41. IEEE 2003.

[11] A. S. Fisher. Computer Aided Software Engineering: Using Software Development Tools. John Wiley & Sons Inc 1991.

[12] P.-A. Muller, F. Fleurey and J.-M. Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In Model Driven Engineering Languages and Systems, LNCS volume 3173, pages 264-278. Springer 2005.

[13] J.-M. Favre. Towards a Basic Theory to Model Model Driven Engineering. In Proceedings of the Third Workshop in Software Model Engineering (WISME). 2004.

[14] D. I. Seidman and J. J. Ritsko. Preface (on Model-Driven Development). In IBM Systems Journal, volume 45, number 3, pages 449-450. IBM 2006.

[15] H. Vangheluwe. Promises and Challenges of Model-Driven Engineering. In Proceedings of the 15th European Conference on Software Maintenance and Reengineering. IEEE 2011.

[16] R. Van Der Straeten, T. Mens and S. Van Baelen. Challenges in Model-Driven Software Engineering. In Models in Software Engineering, Models 2008 Workshops, LNCS volume 5421, pages 35-47. Springer 2009.

[17] P. Mohagheghi, M. A. Fernandez, J. A. Martell, M. Fritzsche and W. Gilani. MDE Adoptation in Industry: Challenges and Success Criteria. In Models in Software Engineering, Models 2008 Workshops, LNCS volume 5421, pages 54-59. Springer 2009.

[18] D. Di Ruscio, L. Iovino and A. Pierantonio. What is Needed for Managing Co-evolution in MDE? In Proceedings of the 2nd International Workshop on Model Comparison in Practice (IWMCP'11), pages 30-38. ACM 2011.

[19] K. S. F. Oliveira and T. C. de Oliveira. A Guidance for Model Composition. In Proceedings of the International Conference on Software Engineering Advances (ICSEA'07), pages 27-32. IEEE 2007.

[20] C. Jeanneret, R. France and B. Baudry. A Reference Process for Model Composition. In Proceedings of the AOSD Workshop on Aspect-Oriented Modeling (AOM'08), pages 1-6. ACM 2008.

[21] R. France, I. Ray, G. Georg and S. Ghosh. Aspect-Oriented Approach to Early Design Modelling. In Software, IEE Proceedings, volume 151, issue 4, pages 173-185. IEE 2004.

[22] M. Didonet Del Fabro, J. Bézivin and P. Valduriez. Weaving Models with the Eclipse AMW Plugin. In Proceedings of the Eclipse Summit Europe (Eclipse Modeling Symposium). 2006.

[23] B. Morin, G. Perrouin, P. Lahire, O. Barais, G. Vanwormhoudt and J.-M. Jézéquel. Weaving Variability into Domain Metamodels. In Software & Systems Modeling, volume 11, issue 3, pages 361-383. Springer 2012.

[24] C. Herrmann, H. Krahn, B. Rumpe, M. Schindler and S. Völkel. An Algebraic View on the Semantics of Model Composition. In Proceedings of the Third ECMDA-FA Conference, Model Driven Architecture - Foundations and Applications, LNCS volume 4530, pages 99-113. Springer 2007.

[25] Y.R. Reddy, S. Ghosh, R. B. France, G. Straw, J.M. Bieman, N. McEachen, E. Song and G. Georg. Directives for Composing Aspect-Oriented Design Class Models. In Transactions on Aspect-Oriented Software Development, LNCS volume 3880, pages 75-105. Springer 2006.

[26] F. Fleurey, B. Baudry, R. France and S. Ghosh A Generic Approach for Automatic Model Composition. In Models in Software Engineering, LNCS volume 5002, pages 7-15. Springer 2008.

[27] A. Abouzahra, J. Bézivin, M. Didonet Del Fabro and F. Jouault. A Practical Approach to Bridging Domain Specific Languages with UML Profiles. In Proceedings of the OOPSLA Workshop on Best Practices for Model Driven Software Development. 2005.

[28] M. Wimmer, A. Schauerhuber, M. Strommer, W. Schwinger and G. Kappel. A Semi-Automatic Approach for Bridging DSLs with UML. In Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM), pages 97-104. 2007.

[29] R. France, F. Fleurey, R. Reddy, B. Baudry and S. Ghosh. Providing Support for Model Composition in Metamodels. In Proceedings of the 11th Enterprise Distributed Object Computing Conference (EDOC 2007). IEEE 2007.

[30] D. S. Kolovos, R. F. Paige and F. A. C. Polack. Merging Models with the Epsilon Merging Language (EML). In Model Driven Engineering Languages and Systems, LNCS volume 4199, pages 215-229. Springer 2006.

[31] I. Groher and M. Voelter. XWeave: Models and Aspects in Concert. In Proceedings of the 10th International Workshop on Aspect-Oriented Modeling (AOM'07), pages 35-40. ACM 2007.

[32] J. Bézivin, S. Bouzitouna, M. Didonet Del Fabro, M.-P. Gervais, F. Jouault, D. Kolovos, I. Kurtev and R. F. Paige. A Canonical Scheme for Model Composition. In Model Driven Architecture - Foundations and Applications, LNCS volume 4066, pages 346-360. Springer 2006.

[33] B. Baudry, F. Fleurey, R. France and R. Reddy. Exploring the Relationship between Model Composition and Model Transformation. In Proceedings of the 7th International Workshop on Aspect Oriented Modeling. 2005.

[34] O. Barais, J Klein, B. Baudry, A. Jackson and S. Clarke. Composing Multi-View Aspect Models. In Proceedings of the 7th International Conference on Composition-Based Software Systems (ICCBSS'08), pages 43-52. IEEE 2008.

[35] P. Sánchez, L. Fuentes, D. Stein, S. Hanenberg and R. Unland. Aspect-Oriented Model Weaving Beyond Model Composition and Model Transformation. In Model Driven Engineering Languages and Systems. LNCS volume 5301, pages 766-781. Springer 2008.

[36] J. Estublier, G. Vega and A. D. Ionita. Composing Domain-Specific Languages for Wide-Scope Software Engineering Applications. In Model Driven Engineering Languages and Systems, LNCS volume 3713, pages 69-83. Springer 2005.

[37] L. Pedro, M. Risoldi, D. Buchs and V. Amaral. Developing Domain-Specific Modeling Languages by Metamodel Semantic Enrichment and Composition: a Case Study In Proceedings of the 10th Workshop on Domain-Specific Modeling (DSM'10). ACM 2010.

[38] B. Morin, J. Klein, O. Barais and J.-M. Jézéquel. A Generic Weaver for Supporting Product Lines. In Proceedings of the 13th International Workshop on Early Aspects (EA'08), pages 11-18. ACM 2008.

[39] M. Bräuer and H. Lochmann. Towards Semantic Integration of Multiple Domain-Specific Languages Using Ontological Foundations. In Proceedings of the 4th International Workshop on Language Enginering (ATEM 3007). 2007.

[40] J. Whittle, P. Jayaraman, A. Elkhodary, A. Moreira and J. Araújo. MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. In Transactions on Aspect-Oriented Software Development VI, LNCS volume 5560, pages 191-237. Springer 2009.

[41] T. Reiter, E. Kapsammer, W. Retschitzegger and W. Schwinger. Model Integration Through Mega Operations. In Proceedings of the International Workshop on Model-Driven Web Engineering (MDWE). 2005.

[42] J. de Lara and E. Guerra. From Types to Type Requirements: Genericity for Model-Driven Engineering. In Software & Systems Modeling, volume 12, issue 3, pages 453-474. Springer 2011.

[43] G. Kappel, E. Kapsammer, H. Kargl, G. Kramler, T. Reiter, W. Retschitzegger, W. Schwinger and M. Wimmer. Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages. In Model Driven Engineering Languages and Systems, LNCS volume 4199, pages 528-542. Springer 2006.

[44] M. K. Hamiaz, M. Pantel, B. Combemale and X. Thirioux. A Formal Framework to Prove the Correctness of Model Driven Engineering Composition Operators.

In Formal Methods and Software Engineering, LNCS volume 8829, pages 235-250. Springer 2014.

[45] J. Oldevik, M. Menarini and I. Krüger. Model Composition Contracts. In Model Driven Engineering Languages and Systems, LNCS volume 5795, pages 531-545. Springer 2009.

[46] J. Kienzle, W. A. Abed, F. Fleurey, J.-M. Jézéquel and J. Klein. Aspect-Oriented Design with Reusable Aspect Models. In Transactions on Aspect-Oriented Software Development VII, LNCS volume 6210, pages 272-320. Springer 2010.

[47] M. Derakhshanmanesh, J. Ebert, T. Iguchi and G. Engels. Model-Integrating Software Components. In Model-Driven Engineering Languages and Systems, LNCS volume 8767, pages 386-402. Springer 2014.

[48] F. Noyrit, S. Gérard and F. Terrier. Computer Assisted Integration of Domain-Specific Modeling Languages Using Text Analysis Techniques. In Model-Driven Engineering Languages and Systems, LNCS volume 8107, pages 505-521. Springer 2013.

[49] E. Vacchi, W. Cazzola, S. Pillay and B. Combemale. Variability Support in Domain-Specific Language Development. In Software Language Engineering, LNCS volume 8225, pages 76-95. Springer 2013.

[50] A. Carton, C. Driver, A. Jackson and S. Clarke. Model-Driven Theme/UML. In Transactions on Aspect-Oriented Development VI, LNCS volume 5560, pages 238-266. Springer 2009.

[51] H. Gomaa and M. E. Shin. A Multiple-View Meta-modeling Approach for Variability Management in Software Product Lines. In Software Reuse: Methods, Techniques, and Tools, LNCS volume 3107, pages 274-285. Springer 2004.

[52] I. Groher and M. Voelter. Aspect-Oriented Model-Driven Software Product Line Engineering. In Transactions on Aspect-Oriented Software Development VI, LNCS volume 5560, pages 111-152. Springer 2009.

[53] J. Steel and J.-M. Jézéquel. On Model Typing. In Software & Systems Modeling, volume 6, issue 4, pages 401-413. Springer 2007.

[54] C. Guy, B. Combemale, S. Derrien, J. R. H. Steel and J.-M. Jézéquel. On Model Subtyping. In Modelling Foundations and Adaptations, LNCS volume 7349, pages 400-415. Springer 2012.

[55] M. Didonet Del Fabro and P. Valduriez. Semi-automatic Model Integration using Matching Transformations and Weaving Models. In Proceedings of the

2007 ACM Symposium on Applied Computing (SAC'07), pages 963-970. ACM 2007.

[56] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus and F. Fouquet. Mashup of Meta-Languages and its Implementation in the Kermeta Language Workbench. In Software & Systems Modeling, volume 14, issue 2, pages 905-920. Springer 2013.

[57] A. Cicchetti, D. Di Ruscio, R. Eramo and A. Pierantonio. Automating Co-evolution in Model-Driven Engineering. In Proceedings of the 12th International Enterprise Distributed Object Computing Conference EDOC'08, pages 222-231. IEEE 2008.

[58] G. Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In ECOOP 2007 - Object-Oriented Programming, LNCS volume 4609, pages 600-624. Springer 2007.

[59] K. Garcés, F. Jouault, P. Cointe and J. Bézivin. Managing Model Adaptation by Precise Detection of Metamodel Changes. In Model Driven Architecture - Foundations and Applications, LNCS volume 5562, pages 34-49. Springer 2009.

[60] A. Narayanan, T. Levendovszky, D. Balasubramanian and G. Karsai. Automatic Domain Model Migration to Manage Metamodel Evolution. In Model Driven Engineering Languages and Systems, LNCS volume 5795, pages 706-711. Springer 2009.

[61] A. Demuth, R. E. Lopez-Herrejon and A. Egyed. Co-evolution of Metamodels and Models through Consistent Change Propagation. In Journal of Systems and Software, issue 111, pages 281-297. Elsevier 2016.

[62] G. Taentzer, F. Mantz, T. Arendt and Y. Lamo. Customizable Model Migration Schemes for Meta-model Evolutions with Multiplicity Changes. In Model-Driven Engineering Languages and Systems, LNCS volume 8107, pages 254-270. Springer 2013.

[63] M. Herrmannsdoerfer, S. Benz and E. Juergens. COPE - Automating Coupled Evolution of Metamodels and Models. In ECOOP 2009 - Object-Oriented Programming, LNCS volume 5653, pages 52-76. Springer 2009.

[64] M. Herrmannsdoerfer, S. D. Vermolen and G. Wachsmuth. An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In Software Language Engineering, LNCS volume 6563, pages 163-182. Springer 2011.

[65] A. Demuth, R. E. Lopez-Herrejon and A. Egyed. Supporting the Co-evolution of Metamodels and Constraints through Incremental Constraint Management. In

Model-Driven Engineering Languages and Systems, LNCS volume 8107, pages 287-303. Springer 2013.

[66] L. M. Rose, D. S. Kolovos, R. F. Paige, F. A. C. Polack and S. Poulding. Epsilon Flock: A Model Migration Language. In Software & Systems Modeling, volume 13, issue 2, pages 735-755. Springer 2012.

[67] T. Arendt, E. Biermann, S. Jurack, C. Krause and G. Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In Model Driven Engineering Languages and Systems, LNCS volume 6394, pages 121-135. Springer 2010.

[68] F. Mantz, A. Rutle, Y. Lamo, A. Rossini and U. Wolter. Towards a Formal Approach to Metamodel Evolution. In Nordic Workshop on Programming Theory. 2010.

[69] C. Krause, J. Dyck and H. Giese. Metamodel-Specific Coupled Evolution Based on Dynamically Typed Graph Transformations. In Theory and Practice of Model Transformations, LNCS volume 7909, pages 76-91. Springer 2013.

[70] A. Cicchetti, D. Di Ruscio and A. Pierantonio. Managing Dependent Changes in Coupled Evolution. In Theory and Practice of Model Transformations, LNCS volume 5563, pages 35-51. Springer 2009.

[71] M. Herrmannsdoerfer, S. Benz and E. Juergens. Automatability of Coupled Evolution of Metamodels and Models in Practice. In Model Driven Engineering Languages and Systems, LNCS volume 5301, pages 645-659. Springer 2008.

[72] B. Gruschko, D. S. Kolovos and R. F. Paige. Towards Synchronizing Models with Evolving Metamodels. In Proceedings of the International Workshop on Model-Driven Software Evolution (MODSE). 2007.

[73] A. Cicchetti, D. Di Ruscio, R. Eramo and A. Pierantonio. Meta-model Differences for Supporting Model Co-evolution. In Proceedings of the 2nd Workshop on Model-Driven Software Evolution. 2008.

[74] A. Cicchetti, D. Di Ruscio and A. Pierantonio. A Metamodel Independent Approach to Difference Representation. In Journal of Object Technology, volume 6, number 9, pages 165-185. 2007.

[75] L. M. Rose, M. Herrmannsdoerfer, J. R. Williams, D. S. Kolovos, K. Garcés, R. F. Paige and F. A. C. Polack. A Comparison of Model Migration Tools. In Model Driven Engineering Languages and Systems, LNCS volume 6394, pages 61-75. Springer 2010.

[76] L. M. Rose, R. F. Paige, D. S. Kolovos and F. A. C. Polack. An Analysis of Approaches to Model Migration. In Proceedings of the Models and Evolution Workshop (MoDSE-MCCM). 2009.

[77] J. C. Mitchell. Concepts in Programming Languages. Cambridge University Press 2002. ISBN 978-0-521-78098-8

[78] J. Bézivin, F. Jouault and P. Valduriez. On the Need for Megamodels. In OOPSLA/GPCE Workshop on Best Practices for Model Driven Software Development. 2004.

[79] E. Ernst. Family Polymorphism. In ECOOP 2001 - Object-Oriented Programming, LNCS volume 2072, pages 303-326. Springer 2001.

[80] R. France and B. Rumpe. Domain Specific Modeling. In Software & Systems Modeling, volume 4, issue 1, pages 1-3. Springer 2005.

[81] J. P. van Gigch. System Design Modeling and Metamodeling. Springer 1991. ISBN 978-0-306-43740-3

[82] T. Mens and S. Demeyer. Software Evolution. Springer 2008. ISBN 978-3-540-76440-3

[83] P. Hudak. Modular Domain Specific Languages and Tools. In Proceedings of the 5th International Conference on Software Reuse (ICSR '98), pages 134-142. IEEE 1998.

[84] E. Vacchi and W. Cazzola. Neverlang: A Framework for Feature-Oriented Language Development. In Computer Languages, Systems & Structure, volume 43, pages 1-40. Elsevier 2015.

[85] F. Jouault and J. Bézivin. KM3: a DSL for Metamodel Specification. In Formal Methods for Open Object-Based Distributed Systems, LNCS volume 4037, pages 171-185. Springer 2006.

[86] D. Steinberg, F. Budinsky, M. Paternostro and E. Merks. EMF: Eclipse Modeling Framework (Second Edition) Addison-Wesley 2008. ISBN 978-0321331885

[87] A. Jackson, J. Klein, B. Baudry and S. Clarke. Executable Aspect Oriented Models for Improved Model Testing. In ECMDA Workshop on Integration of Model Driven Development and Model Driven Testing. 2006.

[88] L. Iovino, A. Pierantonio and I. Malavolta. On the Impact Significance of Metamodel Evolution in MDE. In Journal of Object Technology, volume 11, number 3, pages 1-33. 2012.

[89] P. Mohagheghi, W. Gilani, A. Stefanescu, M. A. Fernandez, B. Nordmoen and M. Fritzsche. Where Does Model-Driven Engineering Help? Experiences from Three Industrial Cases. In Software & Systems Modeling, volume 12, issue 3, pages 619-639. Springer 2011.

[90] J. E. Rivera, J. R. Romero and A. Vallecillo. Behavior. Time and Viewpoint Consistency: Three Challenges for MDE. In Models in Software Engineering, LNCS volume 5421, pages 60-65. Springer 2009.

[91] J. Sprinkle, B. Rumpe, H. Vangheluwe and G. Karsai. Metamodelling, State of the Art and Research Challenges. In Model-Based Engineering of Embedded Real-Time Systems, LNCS volume 6100, pages 57-76. Springer 2010.

[92] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, M. Wimmer and G. Kappel. A Survey on Aspect-Oriented Modeling Approaches. In Technical Report, Vienna University of Technology. 2006.

[93] T. Mens, G. Taentzer and D. Müller. Challenges in Model Refactoring. In Proceedings of the 8th International Workshop on Principles of Sofware Evolution, pages 13-22. IEEE 2005.

[94] J. R. Romero and A. Vallecillo. Well-formed Rules for Viewpoint Correspondences Specification. In Proceedings of the 5th Workshop on Open Distributed Processing for Enterprise Computing (WODPEC 2008). IEEE 2008.

[95] S. Krogdahl, B. Møller-Pedersen and F. Sørensen. Exploring the use of Package Templates for Flexible Re-use of Collections of Related Classes. In Journal of Object Technology, volume 8, number 7, 2005.

[96] E. W. Axelsen, F. Sørensen, S. Krogdahl and B. Møller-Pedersen. Challenges in the Design of the Package Template Mechanism. In Transactions on AOSD, LNCS volume 7271, pages 268-305. Springer 2012.

[97] F. Sørensen, E. W. Axelsen and S. Krogdahl. Reuse and Combination with Package Templates. In Proceedings of the 4th Workshop on Mechanisms for Specialization, Generalization and Inheritance. ACM 2010.

[98] E. W. Axelsen and S. Krogdahl. Package Templates: A Definition by Semantics-Preserving Source-to-Source Transformations to Efficient Java Code. In Proceedings of the 11th International Conference on Generative Programming and Component Engineering (GPCE'12). ACM 2012.

[99] M. Emerson and J. Sztipanovits. Techniques for Metamodel Composition. In Proceedings of the 6th Workshop on Domain Specific Modeling (at OOPSLA), pages 123-139. 2006.

[100] D. Jurafsky and J. H. Martin. Speech and Language Processing (Second Edition). Pearson Education International. 2009.

[101] F. R. Golra, A. Beugnard, F. Dagnat, S. Guerin and C. Guychard. Addressing Modularity for Heterogeneous Multi-model Systems Using Model Federation. In Proceedings of the 15th International Conference on Modularity, pages 206-211. ACM 2016.

[102] M. Herrmannsdoerfer. GMF: A Model Migration Case for the Transformation Tool Contest. In Electronic Proceedings in Theoretical Computer Science (EPTCS), volume 74, pages 1-5. 2011.

[103] R. F. Paige, P. J Brooke and J. S. Ostroff. Metamodel-Based Model Conformance and Multiview Consistency Checking. In ACM Transactions on Software Engineering and Methodology, volume 16, issue 3. ACM 2007.

[104] A. Kalnins and J. Barzdins. Metamodel Specialization for Graphical Modeling Language Support. In Proceedings of the 19th International Conference on Model Driven Engineering Languages and Systems (MODELS '16), pages 103-112. ACM 2016.

[105] A. Kalnins and J. Barzdins. Metamodel Specialization for DSL Tool Building. In Proceedings of the International Baltic Conference on Databases and Information Systems, CCIS volume 615, pages 68-82. Springer 2016.

[106] H. Bruneliere, J. G. Perez, M. Wimmer and J. Cabot. EMF Views: A View Mechanism for Integrating Heterogeneous Models. In Proceedings of the 34th International Conference on Conceptual Modeling (ER 2015), LNCS volume 9381, pages 317-325. Springer 2015.

[107] G. D. Crnkovic. Scientific Methods in Computer Science. In Proceedings of the Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden. 2002.

[108] H. Bruneliere, J. Garcia, P. Desfray, D. E. Khelladi, R. Hebig, R. Bendraou and J. Cabot. On Lightweight Metamodel Extension to Support Modeling Tools Agility. In Modelling Foundations and Applicatons, LNCS volume 9153, pages 62-74. Springer 2015.

[109] S. Sen, N. Moha, B. Baudry and J.-M. Jézéquel. Meta-model Pruning. In Model Driven Engineering Languages and Systems, LNCS volume 5795, pages 32-46. Springer 2009.

[110] H. König and Z. Diskin. Advanced Local Checking of Global Consistency in Heterogeneous Multimodeling. In Modelling Foundations and Applications, LNCS volume 9764, pages 19-35. Springer 2016.

[111] D. Di Ruscio, J. Etzlstorfer, L. Iovino, A. Pierantonio and W. Schwinger. Supporting Variability Exploration and Resolution During Model Migration. In Modelling Foundations and Applications, LNCS volume 9764, pages 231-246. Springer 2016.

[112] W. Kessentini, H. Sahraoui and M. Wimmer. Automated Metamodel/Model Co-evolution Using a Multi-objective Optimization Approach. In Modelling Foundations and Applications, LNCS volume 9764, pages 138-155. Springer 2016.

[113] J. F. Roddick. Schema Evolution in Database Systems - An Annotated Bibliography. In ACM SIGMOD Record, volume 21, issue 4, pages 35-40. ACM 1992.

[114] B. Hoisl, Z. Hu and S. Hidaka. Towards Bidirectional Higher-Order Transformation for Model-Driven Co-evolution. In Model-Driven Engineering and Software Development, CCIS volume 506, pages 153-167. Springer 2015.

[115] A. Haber, M. Look, P. M. S. Nazari, A. N. Perez, B. Rumpe, S. Völkel and A. Wortmann. Composition of Heterogeneous Modeling Languages. In Model-Driven Engineering and Software Development, CCIS volume 580, pages 45-66. Springer 2015.

[116] S. Demeyer. Research Methods in Computer Science. In Proceedings of the 27th International Conference on Software Maintenance (ICSM 2011). IEEE 2011.

[117] A. Etien and C. Salinesi. Managing Requirements in a Co-evolution Context. In Proceedings of the 13th International Conference on Requirements Engineering (RE'05). IEEE 2005.

[118] N. Moha, V. Mahé, O. Barais and J.-M. Jézéquel. Generic Model Refactorings. In Model Driven Engineering Languages and Systems, LNCS volume 5795, pages 628-643. Springer 2009.

Part II

Research Papers

Appendix A

Paper I: Advancing Generic Metamodels

Henning Berg, Birger Møller-Pedersen and Stein Krogdahl

In proceedings of the co-located workshops on SPLASH '11: Domain-Specific Modeling (DSM'11), pages 19-24. ACM 2011.

(The paper has been reformatted to fit the style of this thesis.)

Appendix B

Paper II: Type-Safe Symmetric Composition of Metamodels using Templates

Henning Berg and Birger Møller-Pedersen

In proceedings of the 7th International Workshop on System Analysis and Modeling (SAM 2012), System Analysis and Modeling: Theory and Practice, Lecture Notes in Computer Science (LNCS), volume 7744, pages 160-178. Springer 2013.

Appendix C

Paper III: Specialisation of Metamodels using Metamodel Types

Henning Berg and Birger Møller-Pedersen

In proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2014), Model-Driven Engineering and Software Development, Communications in Computer and Information Science (CCIS), volume 506, pages 84-99. Springer 2015.

The paper was nominated for the Best Student Paper Award.

Appendix D

Paper IV: A Framework for Metamodel Composition and Adaptation with Conformance-Preserving Model Migration

Ingrid Chieh Yu and Henning Berg

In proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2015), Model-Driven Engineering and Software Development, Communications in Computer and Information Science (CCIS), volume 580, pages 133-154. Springer 2015.

Appendix E

Paper V: Metamodel and Model Composition by Integration of Operational Semantics

Henning Berg and Birger Møller-Pedersen

In proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2015), Model-Driven Engineering and Software Development, Communications in Computer and Information Science (CCIS), volume 580, pages 172-189. Springer 2015.

The paper won the Best Student Paper Award.

Appendix F

Paper VI: Migration of Models using Derived Transformations from Metamodel Adaptation and Composition Directives

Henning Berg

Unpublished. 2017.