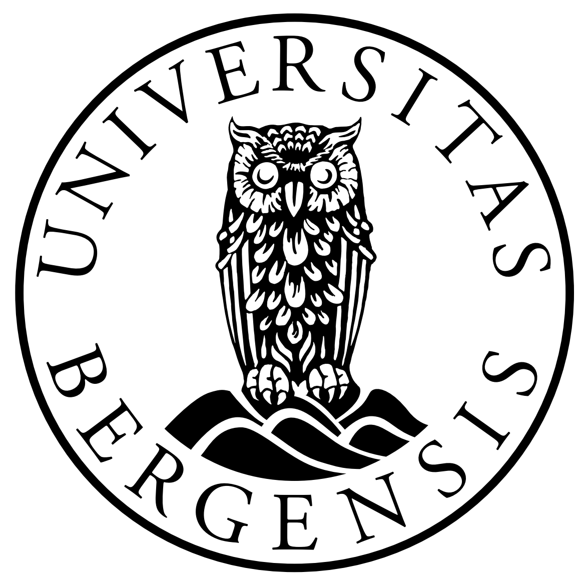
Diagram Predicate Framework meets Model Versioning and Deep Metamodelling

Diagram Predicate Framework meets Model Versioning and Deep Metamodelling

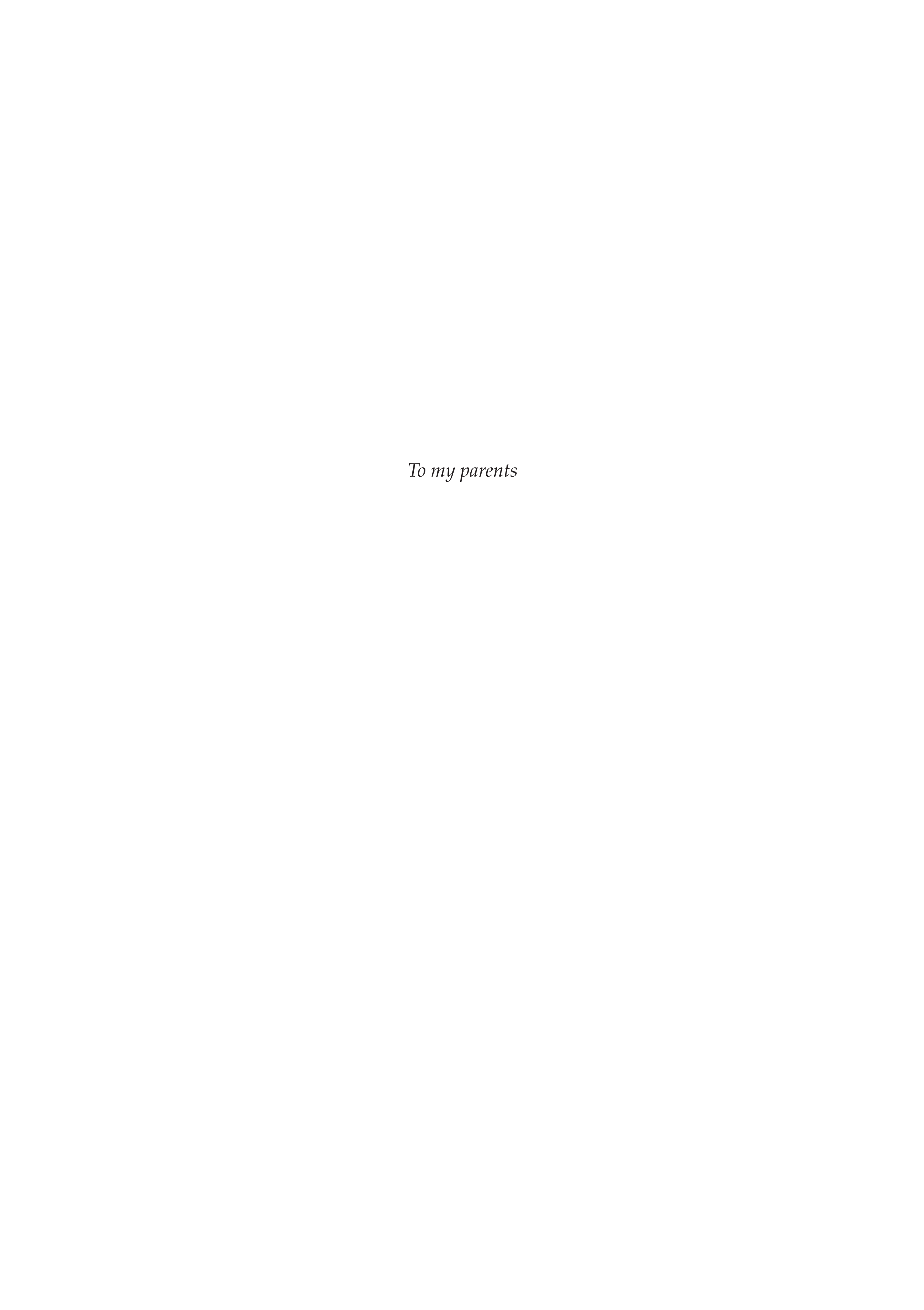
Alessandro Rossini



Dissertation for the degree of Philosophiae Doctor (PhD)
Department of Informatics
University of Bergen

December 2011

ISBN 978-82-308-1900-5
University of Bergen, Norway
Submitted 3rd October 2011
All text and figures © 2011 Alessandro Rossini



To my parents

Contents

Preface ix

Scientific Environment xiii

Abstract xv

1 Model-Driven Engineering 1
- 1.1 Introduction 1
- 1.2 Diagrammatic modelling 2
- 1.3 Metamodelling 5
- 1.4 Constraints 6
- 1.5 Typing and conformance 6

2 Diagram Predicate Framework 9
- 2.1 Graph and graph homomorphism 9
- 2.2 Signature and specification 13
- 2.3 Typing and conformance 22
- 2.4 Specification morphism 22
- 2.5 Specification transformation 24
- 2.6 Specification entailment 27
- 2.7 Related work 32
- 2.8 Conclusion and future work 34

3 Constraint-Aware Model Versioning 35
- 3.1 Introduction 35
- 3.2 Model versioning 36
- 3.3 Calculation and representation of differences 39
- 3.4 Synchronisation 48
  - 3.4.1 Construct the common of commons 50
  - 3.4.2 Construct the difference specifications 53
  - 3.4.3 Construct the merge of differences 53
  - 3.4.4 Detect conflicts 54
  - 3.4.5 Resolve conflicts 60

3.4.6 Construct the synchronised specifications 67
3.5 Related work 71
3.6 Conclusion and future work 74

4 Deep Metamodelling 75
4.1 Introduction 75
4.2 Metamodelling 76
4.3 Deep metamodelling 81
4.3.1 Deep characterisation 81
4.3.2 Double typing and linguistic extension 82
4.3.3 Some open questions in deep metamodelling 85
4.4 Formalisation of deep metamodelling 87
4.4.1 Double metamodelling stack 87
4.4.2 Partial double metamodelling stack 92
4.4.3 Deep metamodelling stack 95
4.5 Flattening of a deep metamodelling stack 109
4.6 Related work 118
4.7 Conclusion and future work 120

5 Conclusion 121

A Appendix 125

Bibliography 129

Preface

The last four years of my life have been dedicated to writing this thesis and to making it as perfect as possible. These years have witnessed days and nights of hard work, discussion, stress, frustration, anguish, insomnia, as well as praise, relief, travelling and fun.

If you are going to read this thesis, I hope that you will find it interesting. If you are just going to browse through it quickly, I hope that you will find the models as beautiful as I do. If you are only interested in this preface, I hope it will leave you with a nice memory.

Bergen, 3rd October 2011

Acknowledgements

This thesis would not have been possible without the contribution of the outstanding individuals I have met during these four years.

First of all, I would like to thank my supervisor Uwe Wolter, for teaching me a lot of interesting knowledge which spans from mathematics to philosophy and history, as well as for giving me invaluable feedback about my research. He deserves much of the credit for this thesis and I am indebted to him for all his help and inspiration, scientifically and otherwise. I would also like to thank my co-supervisor Khalid A. Mughal, for suggesting that I enrol in a PhD programme and for supporting all my choices when I finally followed his suggestion. With time I realised that his initiative saved me from becoming a frustrated software engineer.

A special thanks goes to Adrian Rutle, for helping me to get started with my research and for sharing many good times with me, both in Bergen and while travelling. He has been a brilliant colleague and a good friend, and I have many good memories from these years.

I am grateful to my parents Pompilio and Loretta, for all they have done for me, especially for setting my life on what I believe is the right path. I hope that this thesis will make them as proud of me as I am of them.

"Tusen takk" to Synnøve Solberg Tokerud, for her love and friendship, for teaching me about Norwegian and Norway, as well as for her beautiful smile which always helped me to stay positive.

The Department of Informatics at the University of Bergen has given me a private office, a good salary and great financial support, and I am thankful for that. I would like to thank the Programming Theory group, especially Marc Bezem, Torill Hamre, Anya Helene Bagge, Valentin David, Dag Hovland and Federico Mancini, for creating a stimulating environment to work in, for all the chats about informatics and teaching, for all the empirical studies on espresso and on chocolate spreads, as well as for all the feedback they gave me about my work. I am also grateful to the administration of the Department of Informatics, especially Ida Holen, for patiently listening to my rants every time I needed to vent my frustration, Petter Bjørstad and Torleiv Kløve, for supporting my stays abroad, and Steinar Heldal, for guiding me through the bureaucracy of the University.

My research was carried out in cooperation with fellow researchers from the Department of Computer Engineering at the Bergen University College. Thanks to Yngve Lamo, for his suggestions about how to deal with the Norwegian system, and Florian Mantz, for being an excellent flatmate and for preparing pancakes every Sunday.

Part of this thesis was written during my 4-month stay at the Department of Computer Engineering at the Autonomous University of Madrid. "Muchas gracias" to Juan de Lara and Esther Guerra, for taking care of me during my stay and for giving me plenty of insights which ended up being almost half of this thesis.

I would like to thank my opponents Reiko Heckel and Einar Broch Johnsen, for all the time they have spent reviewing this work, and Michal Walicki, for coordinating the committee. I am also grateful to all my fellow researchers and anonymous reviewers who pointed out flaws and suggested possible improvements in my research.

Despite all the time spent preparing this thesis rather than hanging out, I still have many friends left and they should all be awarded for their patience. In Bergen, Mikal Carlsen Østensen helped me with practically everything before and after my move to Norway. Diego Fiore has been one of my closest friends, who shared countless discussions about the grotesque society we live in with me and was a perfect companion on many suffocating trips around the world. Paolo Angelelli has also been a very good friend, who contributed a lot to the discussion about how to develop an ideal society. My stay in Madrid would not have been the same without Lucia Cammalleri, Teresa Terrana and Daniele Sidoti, who treated me like a close friend since the first day we met. In Italy, my good, old friends Maura Brandimarte, Albert Marsili, Marino Di Carlo, Graziano Liberati and Angelo Di Saverio have been there every time I was back home, and I really appreciate it.

Finally, this thesis would have not reached this level of art without the free and open source software I use and enjoy. A special thanks goes to the communities behind GNU, Linux, KDE, Firefox, Kile, Inkscape, Subversion and Git.

Scientific Environment

The research presented in this thesis has been conducted within the Programming Theory Group of the Department of Informatics at the University of Bergen, as well as within the Department of Computer Engineering at the Autonomous University of Madrid during my 4-month stay.

Abstract

Model-driven engineering (MDE) is a branch of software engineering which aims at improving the productivity, quality and cost-effectiveness of software by shifting the paradigm from code-centric to model-centric. MDE promotes models and modelling languages as the main artefacts of the development process and model transformation as the primary technique to generate (parts of) software systems out of models. Models enable developers to reason at a higher level of abstraction, while model transformation restrains developers from repetitive and error-prone tasks such as coding. Although techniques and tools for MDE have advanced considerably during the last decade, several concepts and standards in MDE are still defined semi-formally, which may not guarantee the degree of precision required by MDE.

This thesis provides a formalisation of concepts in MDE based on the Diagram Predicate Framework (DPF), which was already under development before this work was initiated. DPF is a formal diagrammatic specification framework founded on category theory and graph transformation. In particular, the main contribution of this thesis is the consolidation of DPF and the formalisation of two novel techniques in MDE, namely model versioning and deep metamodelling. The content of this thesis is based on a sequence of publications resulting from the joint work with researchers from the University of Bergen, the Bergen University College and the Autonomous University of Madrid.

The work presented in this thesis is neither purely theoretical nor purely practical; it rather seeks to bridge the gap between these worlds. It provides a formal approach to model versioning and deep metamodelling motivated and illustrated by practical examples, while it introduces only the theoretical constructions which are necessary to investigate, formalise and solve these practical challenges.

This thesis is organised as follows. Chapter 1 introduces MDE along with a discussion regarding some of its fundamental concepts, techniques and standards. Chapter 2 outlines DPF along with a formalisation of some of the fundamental concepts in MDE. In Chapter 3, a formal approach to model versioning is described. In Chapter 4, a formal approach to deep metamodelling is presented. Chapter 5 provides some concluding remarks. Finally, Appendix A details some of the categorical constructions used in this thesis.

CHAPTER 1

Model-Driven Engineering

In this chapter, we introduce MDE along with a discussion regarding some of its fundamental concepts, techniques and standards.

1.1 Introduction

Evolution of programming

Since the beginning of informatics, developing high-quality software at a low cost has been a continuous vision. This vision has boosted several shifts of programming paradigms; e.g., from machine code to assembler programming and from imperative to object-oriented programming. In every shift of paradigm, productivity has been increased by raising the abstraction level of programming languages and techniques. One of the latest steps in this direction has led to the usage of models and modelling languages in development processes.

Model-driven engineering

Initially, models were adopted for mere documentation purposes while source code remained the main artefact of the development process. Lately, however, models have gained a central role in the development process. This trend has led to a branch of software engineering which pursues the shift of paradigm from code-centric to model-centric. In the literature, this branch is referred to as model-driven engineering (MDE), model-driven development (MDD) and model-driven software development (MDSD). In this thesis, we adopt the term MDE to denote this branch.

Advantages of MDE

MDE promotes models as the main artefacts of the development process as well as model transformation as the primary technique to automatically generate (parts of) software systems. By raising the abstraction level from source code, models enable developers and domain experts to focus on the problem domain rather than implementation details. By automating repetitive and error-prone tasks such as coding, model transformation enhances productivity, reusability and quality.

Model-Driven Architecture

The reference industrial standardisation of MDE is the Model-Driven Architecture (MDA), which was initiated by the Object Management Group (OMG) [66] late in 2000 [39, 53, 67, 74]. The basic ideas of MDA are closely related to generative programming [25], software factories [45], domain-specific languages [58], etc. MDA is based on multiple standards, including the Meta-Object Facility (MOF) [68], the Unified Modeling Language (UML) [71], the Object Constraint Language (OCL) [70] and the XML Metadata Interchange (XMI) [69].

Eclipse Modeling Framework

Some popular implementations of the MDA standards exist. The de-facto standard is the Eclipse Modeling Framework (EMF) [32, 86], which is part of the Eclipse project [33].

1.2 Diagrammatic modelling

Model

The term *model* may have different meanings depending on the context. In [20], one of the definitions of model is "a representation of something, either as a physical object which is usually smaller than the real object, or as a simple description of the object which might be used in calculations". In software engineering, a model denotes "an abstraction of a (real or language-based) system allowing predictions or inferences to be made" [55]. In formal specifications such as formal logic and universal algebra, in contrast, a system is represented by a specification, i.e., a set of logical formulae, while a model of such a specification denotes a mathematical structure which satisfies these formulae. Thus, formal specifications correspond to models in terms of software modelling. In this thesis, we interpret the term model from the software engineering perspective.

Descriptive vs. prescriptive

Models are often categorised into *descriptive* and *prescriptive*: a descriptive model describes an existing original, e.g., a map of a real city with streets, buildings, etc. while a prescriptive model specifies aspects of an original which is to be built, e.g., a blueprint of a building. In software engineering, models may be both prescriptive and descriptive: a model can be used to represent relevant aspects of a software system and later on drive the implementation of the same software system (see Figure 1.1).

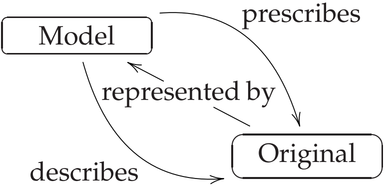


Figure 1.1: A model may describe or prescribe an original

DIAGRAM

The term *diagram* may also have different meanings depending on the context. In [20], one of the definitions of diagram is "a geometric symbolic representation of information according to some visualisation technique", e.g., chart diagrams and cake diagrams. In software engineering, a diagram denotes a structure which is based on graphs, i.e., a set of nodes (or vertices) together with a set of arrows (or edges) between nodes. In category theory, in contrast, a diagram denotes a graph homomorphism from a shape graph into a graph [12, 38]. In this thesis, we interpret the term diagram from the software engineering perspective.

DIAGRAMMATIC VS. VISUAL

Since graph-based structures are often visualised in a natural way, the terms *diagrammatic* and *visual* and are often treated as synonyms. In this thesis, however, we distinguish between these terms. A diagrammatic model denotes a model which is represented by a graph-based structure, while a visual model denotes a model which is intuitive for humans. Although it is feasible to visualise graph-based structures, it may be a challenging task, and sometimes even unfeasible, to find intuitive visualisations for all aspects of diagrammatic models.

HISTORY OF DIAGRAMMATIC MODELS

Diagrammatic models have already been adopted in software engineering for some decades; e.g., flowcharts (Seventies) for the description of behavioural properties of software systems; petrinets (Eighties) for the specification of discrete distributed systems; entity-relationship diagrams (Eighties) for the conceptual modelling of data structures.

CONCEPTUAL TWO-DIMENSIONALITY

A factor which has helped in the popularisation of diagrammatic models is the conceptual two-dimensionality of the modelled universes [31], e.g., nodes and arrows, *activities* and *decisions*, *places* and *transitions*, *entities* and *relations*, *classes* and *associations*, *objects* and *links*, etc. Each of these models may be represented by graph-based structures with nodes representing the first dimension and arrows representing the second dimension of the modelled universe.

UNIFIED MODELING LANGUAGE

Several modelling languages have emerged in the last few years as attempts to facilitate MDE. In the state-of-the-art of MDE, models are often specified by means of MOF-based modelling languages such as UML. UML includes a set of languages which are used to describe or specify *structural* and *behavioural* aspects of object-oriented software systems. The following example illustrates the usage of a *UML class diagram* to represent structural aspect of an object-oriented software system. Note that the example is intentionally kept simple, retaining only the details which are relevant for the discussion.

Example 1 (UML class diagram)**.** *Let us consider an information system for the management of students, universities and projects. This information system should satisfy the following requirements:*

1. *A university educates* none *to* many *students.*
2. *A student studies* at least at one *and* at most at four *universities.*

Figure 1.2 shows a UML class diagram representing an object-oriented structural model of the information system above.

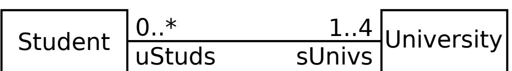


Figure 1.2: A UML class diagram

The class diagram consists of two classes *Student and University and a bidirectional* association *between the classes. The bidirectional association has two* role names *uStuds and sUnivs together with two* multiplicity constraints *0..* and 1..4. The requirements 1 and 2 are enforced in the class diagram by the multiplicity constraints 0..* and 1..4, respectively.*

Instance

The term *instance* may also have different meanings depending on the context. In software engineering, an instance denotes a structure which satisfies the requirements of its corresponding class or, more generally, its corresponding model; i.e., a model restricts the set of its valid instances. The following example illustrates the usage of *UML object diagrams* to represent possible instances of an object-oriented software system at a point in time.

Example 2 (UML object diagram)**.** *Building upon Example 1, Figure 1.3 shows a UML object diagram representing a possible instance of the information system above.*

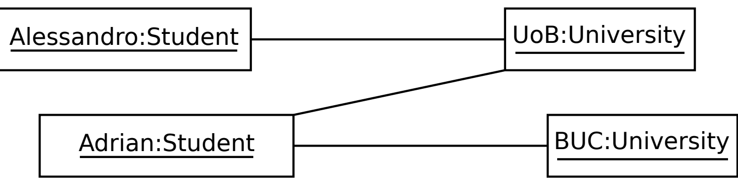


Figure 1.3: A UML object diagram

The UML object diagram consists of two objects *Alessandro and Adrian of type Student, two objects UoB and BUC*[1] *of type University, and three* links *between the objects. Note that these objects are just* illustrations *of possible runtime objects of the considered information system.*

[1] “UoB” and “BUC” stand for University of Bergen and Bergen University College, respectively.

1.3 Metamodelling

Metamodel

The precise definition of the term *metamodel* is frequently debated in the literature (see [9, 16, 17, 43, 48, 55, 56, 85] for a comprehensive discussion). Conceptually, the prefix *meta-* suggests that modelling has occurred twice, which is reflected in the definition "[a metamodel is] a model of models" [67]. Technically, a metamodel defines the abstract syntax of a modelling language. The abstract syntax describes the set of modelling concepts, their attributes and their relationships, as well as the rules for combining these concepts to specify valid models [71]. This means that a metamodel restricts the set of its valid instances in the same way a model does, which is reflected in the definition "a model is an instance of a metamodel" [71]. The following example illustrates the usage of a simplified UML metamodel to represent the modelling concepts of class diagrams.

Example 3 (UML metamodel for class diagrams)**.** *Building upon Example 1, Figure 1.4(a) shows a simplified UML metamodel for class diagrams. Figure 1.4 also shows some of the relations between the class diagram and the metamodel as dashed, grey arrows.*

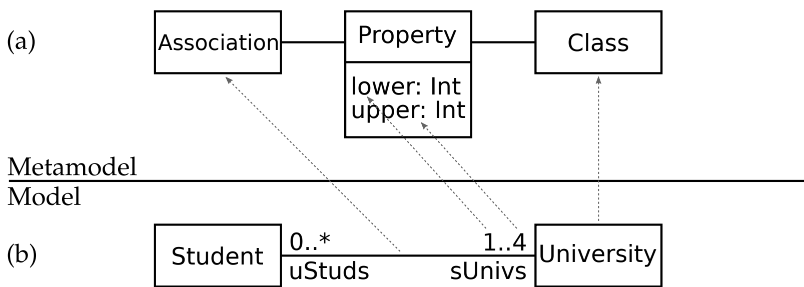


Figure 1.4: A simplified UML metamodel for class diagrams

The metamodel consists of three metaclasses Class*,* Property *and* Association *and two bidirectional associations between the metaclasses. The metaclass* Property *has two* attributes lower *and* upper*. The classes* Student *and* University *in the class diagram are instances of the metaclass* Class *in the metamodel. The multiplicity constraints* 0..* *and* 1..4 *in the class diagram are specified by the attributes* lower *and* upper *of the metaclass* Property *in the metamodel. Note that each model element in a UML class diagram is an instance of exactly one model element in the UML metamodel.*

Note that the UML metamodel, in turn, is a valid instance of the MOF metamodel. UML diagrams, UML and MOF are part of the so-called OMG's 4-layer hierarchy [17], which is described in detail in Section 4.2.

1.4 Constraints

Structural vs. attached constraints

MOF-based modelling languages allow for the specification of simple constraints such as multiplicity and uniqueness constraints, hereafter called *structural constraints*. These structural constraints are usually specified by attributes of classes in the corresponding metamodel of the modelling language. However, these structural constraints may not be sufficient to specify complex system requirements. Hence, metamodels are often complemented with textual constraint languages such as OCL to specify more complex constraints, hereafter called *attached constraints*. The following example illustrates the combination of UML class diagrams with OCL constraints.

Example 4 (UML class diagram with attached OCL constraint)**.** *Let us consider once again the information system of Example 1. This information system is extended with the following additional requirements:*

3. *A project involves* none *to* many *students.*
4. *A project must be controlled by* at least one *university.*
5. *A student involved in a project must study at* at least one *of the controlling universities.*

Figures 1.5(b) and 1.5(c) show a UML class diagram and an attached OCL constraint, respectively, which are compliant with the requirements above.

The requirements 1, 2, 3 and 4 are enforced in the UML class diagram by multiplicity constraints. The requirement 5, however, can only be enforced by an attached OCL constraint.

1.5 Typing and conformance

Typed by and conforms to

In MDE, the terms *typing* and *conformance* and are often used interchangeably. In this thesis, however, we distinguish between these terms. A model is said to be *typed by* a metamodel if each element in the model is typed by an element in the metamodel, while a model is said to *conform to* a metamodel if it is typed by the metamodel and, in addition, satisfies all (structural and attached) constraints of the metamodel.

Linguistic and ontological typing

UML object diagrams and UML class diagrams are located at the same model metalevel although UML object diagrams can be regarded as models which are typed by UML class diagrams. At the same time, since UML object diagrams are at the model metalevel, they are regarded as models which are typed by the UML metamodel. These two flavours of typing are referred to as *ontological* and *linguistic*, respectively [9, 48, 55, 56]. The following example illustrates these two flavours of typing.

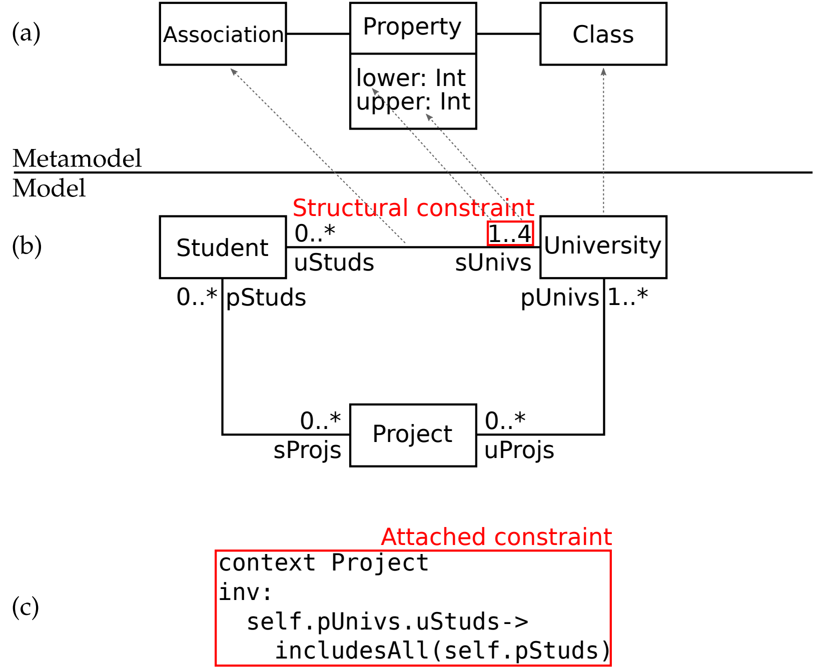


Figure 1.5: A UML class diagram together with an OCL constraint

Example 5 (Linguistic and ontological typing)**.** *Figures 1.6(a) and 1.6(b) show a simplified UML metamodel and a UML class/object diagram, respectively.*

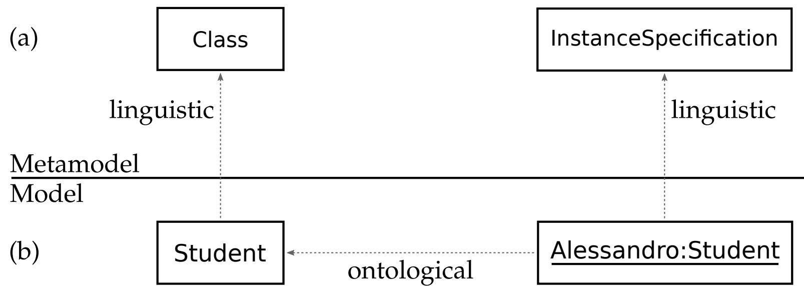


Figure 1.6: Linguistic and ontological instantiation

The class Student is linguistically typed by the metaclass Class, while the object Alessandro is ontologically typed by the class Student and linguistically typed by the metaclass InstanceSpecification.

Note that a model with these two flavours of typing is said to have double linguistic/ontological typing. A metamodelling hierarchy which supports double linguistic/ontological typing is described in detail in Section 4.3.

CHAPTER 2

Diagram Predicate Framework

In the previous chapter, we introduced MDE along with a discussion regarding some of its fundamental concepts. In this chapter, we outline DPF along with a formalisation of some of the fundamental concepts in MDE. DPF will be adopted as the formal underpinning for the formalisation of model versioning and deep metamodelling presented in the following chapters.

2.1 Graph and graph homomorphism

Kinds of graphs

In a first approximation, diagrammatic models can be represented by graphs of different kinds, e.g., simple graphs, bipartite graphs, directed graphs, directed multi-graphs, attributed graphs, hypergraphs, etc. Graphs are a well-known and well-understood means to represent structural and behavioural properties of software systems [36]. In this thesis, we adopt directed multi-graphs.

A directed multi-graph consists of a set of nodes together with a set of arrows, where multiple arrows between the same source and target nodes are permitted. Graphs are related by graph homomorphisms. A graph homomorphism consists of a pair of maps from the nodes and arrows of a graph to those of another graph, where the maps preserve the source and target of each arrow.

Definition 1 (Graph). *A graph $G = (G_N, G_A, src^G, trg^G)$ consists of a set G_N of nodes (or vertices), a set G_A of arrows (or edges) and two maps $src^G, trg^G : G_A \to G_N$ assigning the source and target to each arrow, respectively. $f : X \to Y$ denotes that $src(f) = X$ and $trg(f) = Y$.*

Definition 2 (Subgraph). *A graph $G = (G_N, G_A, src^G, trg^G)$ is subgraph of a graph $H = (H_N, H_A, src^H, trg^H)$, written $G \sqsubseteq H$, if and only if $G_N \subseteq H_N$, $G_A \subseteq H_A$ and $src^G(f) = src^H(f), trg^G(f) = trg^H(f)$, for all $f \in G_A$.*

Definition 3 (Graph homomorphism). *A graph homomorphism $\phi : G \to H$ consists of a pair of maps $\phi_N : G_N \to H_N$, $\phi_A : G_A \to H_A$ which preserve the sources and targets, i.e., for each arrow $f : X \to Y$ in G we have $\phi_A(f) : \phi_N(X) \to \phi_N(Y)$ in H.*

Remark 1 (Inclusion graph homomorphism). *$G \sqsubseteq H$ if and only if the inclusion maps $inc_N : G_N \hookrightarrow H_N$ and $inc_A : G_A \hookrightarrow H_A$ define a graph homomorphism $inc : G \hookrightarrow H$.*

Having defined graphs and graph homomorphisms, it is natural to consider all graphs and graph homomorphisms as objects and morphisms, respectively, of a category [12, 38]. The category of graphs is defined as follows:

Definition 4 (Category of graphs). *The category* **Graph** *has all graphs G as objects and all graph homomorphisms $\phi : G \to H$ as morphisms between graphs G and H.*

The composition $\phi;\psi : G \to K$ of two graph homomorphisms $\phi : G \to H$ and $\psi : H \to K$ is defined component-wise $\phi;\psi = (\phi_N, \phi_A);(\psi_N, \psi_A) := (\phi_N;\psi_N, \phi_A;\psi_A)$. The identity graph homomorphisms $id^G : G \to G$ are also defined component-wise $id^G = (id^{G_N}, id^{G_A})$. This ensures that the composition of graph homomorphisms is associative and that identity graph homomorphisms are left and right neutral with respect to composition.

Semantics of nodes and arrows

The semantics of nodes and arrows of a graph has to be chosen in a way which is appropriate for the corresponding modelling environment [82]. In object-oriented structural modelling, each object may be related to a set of other objects. Hence, it is appropriate to interpret nodes as sets and arrows $\mathsf{X} \xrightarrow{\mathsf{f}} \mathsf{Y}$ as multi-valued functions $f : X \to \wp(Y)$. The powerset $\wp(Y)$ of Y is the set of all subsets of Y, i.e., $\wp(Y) = \{A \mid A \subseteq Y\}$. Moreover, the composition of two multi-valued functions $f : X \to \wp(Y)$, $g : Y \to \wp(Z)$ is defined by $(f;g)(x) := \bigcup\{g(y) \mid y \in f(x)\}$. The following example illustrates the usage of graphs to represent object-oriented structural models.

Example 6 (Graph)**.** *Building upon Example 4, Figure 2.1 shows a graph G representing a simplified object-oriented structural model of the information system above.*

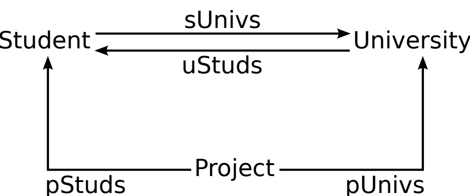


Figure 2.1: A graph G

In G, the nodes Student*,* University *and* Project *are interpreted as sets $Student$, $University$ and $Project$, and the arrows* sUnivs*,* uStuds*,* pUnivs *and* pStuds *are interpreted as multi-valued functions $sUnivs : Student \to \wp(University)$, etc.*

INDEXED SEMANTICS

The semantics of a graph can be formally defined in either an *indexed* or a *fibred* way [31, 90]. In the indexed version, the semantics of a graph is given by all graph homomorphisms $sem : G \to \mathbf{U}$ from the graph G into a category $\mathbf{U}$, e.g., **Set** (sets as objects and functions as morphisms) or **Mult** (sets as objects and multi-valued functions as morphisms as described above).

FIBRED SEMANTICS

In the fibred version, the semantics of a graph is given by the set of its instances. An instance (I, ι) of a graph G consists of a graph I together with a graph homomorphism $\iota : I \to G$. The following example illustrates the usage of graphs and graph homomorphisms to represent instances of a graph.

Example 7 (Instance of graph)**.** *Building upon Example 4, Figure 2.2(b) shows a graph I representing an instance of the graph G. Figure 2.2 also shows some of the mappings of the graph homomorphism $\iota : I \to G$ as dashed, grey arrows.*

The mappings of the nodes of the graph homomorphism ι are defined as follows:

ι(*Alessandro*) = ι(*Adrian*) = *Student*
ι(*UoB*) = ι(*BUC*) = *University*
ι(*DPF*) = ι(*Distech*) = *Project*

The mappings of the arrows of the graph homomorphism ι are defined accordingly.

The graph G alone is not sufficient to capture all the requirements 1, 2, 3, 4 and 5 (see Examples 1 and 4); e.g., the arrow from the node Distech *to the node* Alessandro *in the graph I represents the information "the project Distech involves the student Alessandro", but, according to requirement 5, "the project Distech can not involve the student Alessandro because he is not a student at the Bergen University College".*

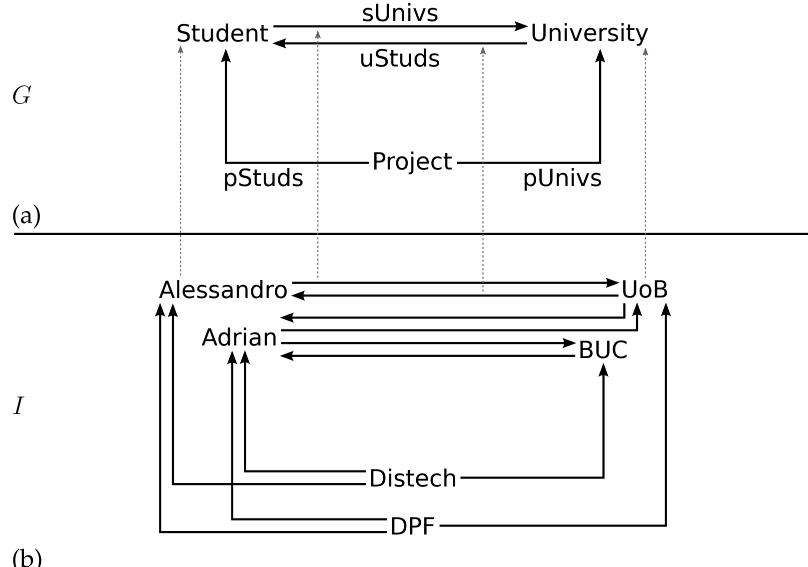


Figure 2.2: The graph G and a possible instance I

Although the usage of graphs for the representation of diagrammatic models is a success story, an enhancement of the formal basis is needed to specify diagrammatic constraints and define a conformance relation between models which takes into account these constraints.

Categorical sketches

A natural choice for this enhancement is category theory, and in particular the categorical sketch formalism, which can be used to define the semantics of diagrams and thus of diagrammatic models. In the categorical sketch formalism, a model is represented by a graph, and properties of the model are expressed by universal properties such as limits, colimits and commutativity constraints [12, 38]. This approach has the benefit of being generic and at a high level of abstraction, but it turns models into a complex categorical structure with several auxiliary objects [31].

Generalised sketches

The proposed formal underpinning of this thesis is the Diagram Predicate Framework (DPF) [78, 79, 80, 81, 82, 83, 84], which is a generalisation and adaptation of the categorical sketch formalism, where the constraining constructs of modelling languages are represented by user-defined signatures in a more intuitive and adequate way. In particular, DPF is an extension of the Generalised Sketches [60] formalism. This extension was originally developed by Diskin et al. in [28, 29, 30].

2.2 Signature and specification

In DPF, a model is represented by a *specification* $\mathfrak{S}$. A specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ consists of an *underlying graph* S together with a set of *atomic constraints* $C^{\mathfrak{S}}$ which are specified by means of a *signature* Σ. A signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$ consists of a set of *predicates* $\pi \in \Pi^{\Sigma}$, each having an arity (or shape graph) $\alpha^{\Sigma}(\pi)$. An atomic constraint (π, δ) consists of a predicate $\pi \in \Pi^{\Sigma}$ together with a graph homomorphism $\delta : \alpha^{\Sigma}(\pi) \to S$ from the arity of the predicate to the underlying graph of the specification.

Definition 5 (Signature)**.** *A signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$ consists of a set of predicate symbols Π^{Σ} and a map α^{Σ} which assigns a graph to each predicate symbol $\pi \in \Pi^{\Sigma}$. $\alpha^{\Sigma}(\pi)$ is called the* arity *of the predicate symbol π.*

Definition 6 (Atomic constraint)**.** *Given a signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$, an atomic constraint (π, δ) on a graph S consists of a predicate symbol $\pi \in \Pi^{\Sigma}$ and a graph homomorphism $\delta : \alpha^{\Sigma}(\pi) \to S$.*

Definition 7 (Specification)**.** *Given a signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$, a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ consists of a graph S and a set $C^{\mathfrak{S}}$ of atomic constraints (π, δ) on S with $\pi \in \Pi^{\Sigma}$.*

The following example illustrates the usage of signatures and specifications to represent object-oriented structural models.

Example 8 (Signature and specification)**.** *Building upon Example 7, Table 2.1 shows a sample signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$ suitable for object-oriented structural modelling. The first column of the table shows the predicate symbols. The second and the third columns show the arities of predicates and a proposed visualisation of the corresponding atomic constraints, respectively. Finally, the fourth column presents the semantic interpretation of each predicate.*

Figure 2.3 shows a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ representing an object-oriented structural model of the information system above.

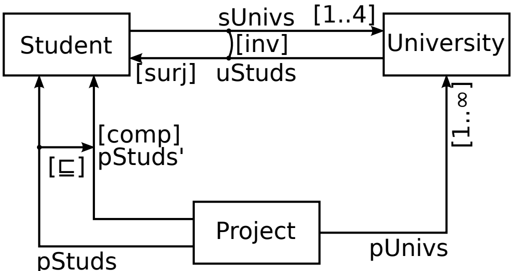


Figure 2.3: A specification $\mathfrak{S}$

Table 2.1: A sample signature Σ

$\pi \in \Pi^{\Sigma}$	$\alpha^{\Sigma}(\pi)$	Proposed vis.	Semantic interpretation
`[mult(m,n)]`	$1 \xrightarrow{a} 2$	X —f [m..n]→ Y	$\forall x \in X : m \leq \lvert f(x) \rvert \leq n$, with $0 \leq m \leq n$ and $n \geq 1$
`[injective]`	$1 \xrightarrow{a} 2$	X —f [inj]→ Y	$\forall x, x' \in X : f(x) = f(x')$ implies $x = x'$
`[surjective]`	$1 \xrightarrow{a} 2$	X —f [surj]→ Y	$\forall y \in Y \, \exists x \in X : y \in f(x)$
`[inverse]`	$1 \underset{b}{\overset{a}{\rightleftarrows}} 2$	X f, g [inv] Y	$\forall x \in X, \forall y \in Y : y \in f(x)$ if and only if $x \in g(y)$
`[irreflexive]`	1 with loop a	X with loop f [irr]	$\forall x \in X : x \notin f(x)$
`[composition]`	$1 \xrightarrow{a} 2$, $2 \xrightarrow{b} 3$, $1 \xrightarrow{c} 3$	X —f→ Y, Y —g→ Z, X —h [comp]→ Z	$\forall x \in X : h(x) = \bigcup\{g(y) \mid y \in f(x)\}$
`[image-inclusion]`	$1 \xrightarrow{a} 2$, $1 \xrightarrow{b} 2$	X f, g [⊑] Y	$\forall x \in X : f(x) \subseteq g(x)$

In $\mathfrak{S}$, the nodes Student*,* University *and* Project *are interpreted as sets Student, University and Project, and the arrows* sUnivs*,* uStuds*,* pUnivs *and* pStuds *are interpreted as multi-valued functions sUnivs : Student $\rightarrow$ $\wp$(University), etc.*

Based on the requirement 2 (see Example 1), the function sUnivs has cardinality between one and four. In $\mathfrak{S}$, this is enforced by the atomic constraint (`[mult(1,4)]`, δ_1) on the arrow sUnivs*. This atomic constraints is formulated by the predicate `[mult(m,n)]` from the signature Σ (see Table 2.1). Moreover, the function uStuds is* surjective*. In $\mathfrak{S}$, this is enforced by the atomic constraint (`[surjective]`, δ_3) on the arrow* uStuds*. Furthermore, the functions sUnivs and uStuds are* inverse *of each other; i.e., $\forall s \in$ Student and $\forall u \in$ University : $s \in uStuds(u)$ if and only if $u \in sUnivs(s)$. In $\mathfrak{S}$, this is enforced by the atomic constraint (`[inverse]`, δ_2) on* sUnivs *and* uStuds*. Finally, based on the requirement 5 (see Example 4), the image of the function pStuds has to be included in the image the composition of the functions pUnivs and uStuds. In $\mathfrak{S}$, this is enforced by the atomic constraints (`[composition]`, δ_4) on the arrows* pUnivs*,* uStuds *and* pStuds'*, and (`[image-inclusion]`, δ_5) on the arrows* pStuds *and* pStuds'*. The graph homomorphisms δ_1, δ_2, δ_3, δ_4 and δ_5 are defined as follows (see Table 2.2):*

$\delta_1(1) = \textsf{Student}$, $\delta_1(2) = \textsf{University}$, $\delta_1(a) = \textsf{sUnivs}$

$\delta_2(1) = \textsf{Student}$, $\delta_2(2) = \textsf{University}$, $\delta_2(a) = \textsf{sUnivs}$, $\delta_2(b) = \textsf{uStuds}$

$\delta_3(1) = \textsf{University}$, $\delta_3(2) = \textsf{Student}$, $\delta_3(a) = \textsf{uStuds}$

$\delta_4(1) = \textsf{Project}$, $\delta_4(2) = \textsf{University}$, $\delta_4(3) = \textsf{Student}$,
$\delta_4(a) = \textsf{pUnivs}$, $\delta_4(b) = \textsf{uStuds}$, $\delta_4(c) = \textsf{pStuds'}$

$\delta_5(1) = \textsf{Project}$, $\delta_5(2) = \textsf{Student}$, $\delta_5(a) = \textsf{pStuds}$, $\delta_5(b) = \textsf{pStuds'}$

Table 2.2: The atomic constraints $(\pi, \delta) \in C^{\mathfrak{S}}$ and their graph homomorphisms

(π, δ)	$\alpha^{\Sigma}(\pi)$	$\delta(\alpha^{\Sigma}(\pi))$
$([\texttt{mult(1,4)}], \delta_1)$	$1 \xrightarrow{a} 2$	Student $\xrightarrow{\text{sUnivs}}$ University
$([\texttt{inverse}], \delta_2)$	$1 \xrightarrow{a} 2$, $2 \xrightarrow{b} 1$	Student $\xrightarrow{\text{sUnivs}}$ University, University $\xrightarrow{\text{uStuds}}$ Student
$([\texttt{surjective}], \delta_3)$	$1 \xrightarrow{a} 2$	University $\xrightarrow{\text{uStuds}}$ Student
$([\texttt{composition}], \delta_4)$	$1 \xrightarrow{a} 2$, $2 \xrightarrow{b} 3$, $1 \xrightarrow{c} 3$	Project $\xrightarrow{\text{pUnivs}}$ University, University $\xrightarrow{\text{uStuds}}$ Student, Project $\xrightarrow{\text{pStuds'}}$ Student
$([\texttt{image-inclusion}], \delta_5)$	$1 \xrightarrow{a} 2$, $1 \xrightarrow{b} 2$	Project $\xrightarrow{\text{pStuds}}$ Student, Project $\xrightarrow{\text{pStuds'}}$ Student

Remark 2 (Predicate symbols)**.** *Some of the predicate symbols in* Σ *(see Table 2.1) refer to single predicates, e.g.,* `[surjective]`*, while some others refer to a family of predicates, e.g.,* `[mult(m,n)]`*. In the case of* `[mult(m,n)]`*, the predicate is parametrised by the (non-negative) integers* m *and* n*, which represent the lower and upper bounds, respectively, of the cardinality of the function which is constrained by this predicate.*

Semantics of predicates

The semantics of predicates of the signature Σ (see Table 2.1) is described using the mathematical language of set theory. In an implementation, the semantics of a predicate is typically given by the code of a corresponding validator such that the mathematical and the validator semantics should coincide. However, it is not necessary to choose between the above mentioned possibilities; it is sufficient to know that any of these possibilities defines valid instances of predicates.

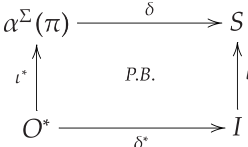
Definition 8 (Semantics of predicates)**.** *Given a signature* $\Sigma = (\Pi^\Sigma, \alpha^\Sigma)$*, a semantic interpretation* $[\![..]\!]^\Sigma$ *of* Σ *consists of a mapping that assigns to each predicate symbol* $\pi \in \Pi^\Sigma$ *a set* $[\![\pi]\!]^\Sigma$ *of graph homomorphisms* $\iota : O \to \alpha^\Sigma(\pi)$*, called valid instances of* π*, where* O *may vary over all graphs.* $[\![\pi]\!]^\Sigma$ *is assumed to be closed under isomorphisms.*

Semantics of a specification

The semantics of a specification is defined in the fibred way [31, 90]; i.e., the semantics of a specification $\mathfrak{S} = (S, C^\mathfrak{S}:\Sigma)$ is given by the set of its instances (I, ι). An instance (I, ι) of a specification $\mathfrak{S}$ consists of a graph I together with a graph homomorphism $\iota : I \to S$ which satisfies the set of atomic constraints $C^\mathfrak{S}$.

To check that an atomic constraint is satisfied in a given instance of a specification $\mathfrak{S}$, it is enough to inspect only the part of $\mathfrak{S}$ which is affected by the atomic constraint. This kind of restriction to a subpart is obtained by the pullback construction [12, 38], which can be regarded as a generalisation of the inverse image construction.

Definition 9 (Instance of a specification)**.** *Given a specification* $\mathfrak{S} = (S, C^\mathfrak{S}: \Sigma)$*, an instance* (I, ι) *of* $\mathfrak{S}$ *consists of a graph* I *and a graph homomorphism* $\iota : I \to S$ *such that for each atomic constraint* $(\pi, \delta) \in C^\mathfrak{S}$ *we have* $\iota^* \in [\![\pi]\!]^\Sigma$*, where the graph homomorphism* $\iota^* : O^* \to \alpha^\Sigma(\pi)$ *is given by the following pullback:*



The following example illustrates the usage of graphs to represent instances of a specification.

Example 9 (Instance of a specification)**.** *Building upon Example 8, Figure 2.4(b) shows a graph* I *representing an instance of the specification* $\mathfrak{S}$*. Figure 2.4 also shows some of the mappings of the graph homomorphism* $\iota : I \to G$ *as dashed, grey arrows.*

The graph homomorphism ι *is defined as in Example 7 and satisfies the set of atomic constraints* $C^\mathfrak{S}$*. If the graph* I *contained an arrow from the node* Distech *to the node* Alessandro *(shown as a dotted, red arrow), it would not be a valid instance of* $\mathfrak{S}$ *since it would violate the atomic constraint (*`[image-inclusion]`*,* δ_5*):*

$pStuds(Distech) = \{Alessandro, Adrian\} \nsubseteq$
$uStuds(pUnivs(Distech)) = \{Adrian\}$

Given a specification $\mathfrak{S}$, the category of instances of $\mathfrak{S}$ is defined as follows:

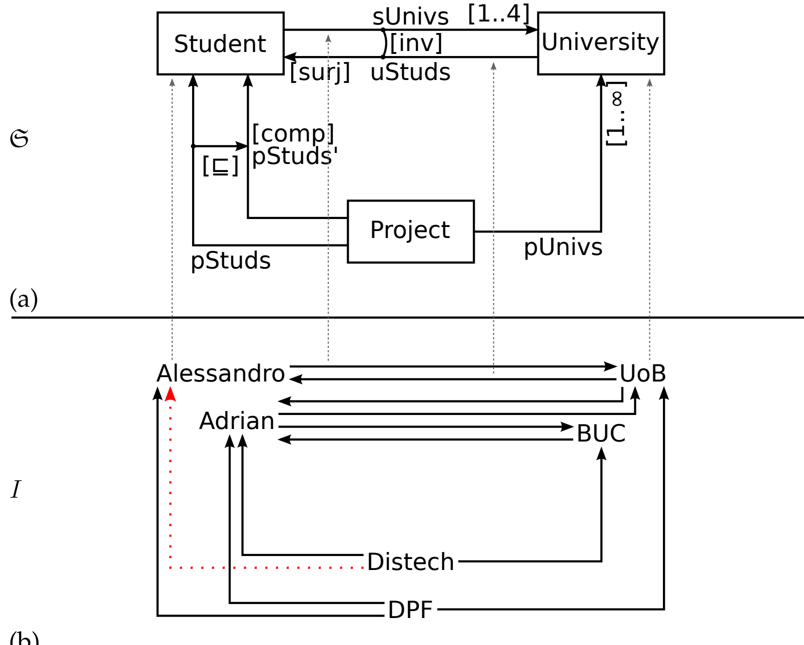
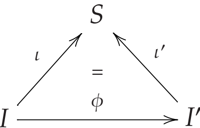


Figure 2.4: The specification $\mathfrak{S}$ and a possible instance I

Definition 10 (Category of instances)**.** *Given a specification* $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$*, the category* $\mathbf{Inst}(\mathfrak{S})$ *has all instances* (I, ι) *of* $\mathfrak{S}$ *as objects and all graph homomorphisms* $\phi : I \to I'$ *as morphisms between instances* (I, ι) *and* (I', ι')*, such that* $\iota = \phi; \iota'$*.*



$\mathbf{Inst}(\mathfrak{S})$ *is a full subcategory of* $\mathbf{Inst}(S)$ *where* $\mathbf{Inst}(S) = (\mathbf{Graph} \downarrow S)$ *is the comma category of all graphs typed by* S *[12]; i.e., we have an inclusion functor* $inc^{\mathfrak{S}} : \mathbf{Inst}(\mathfrak{S}) \hookrightarrow \mathbf{Inst}(S)$*.*

As mentioned, in an implementation, the semantics of a predicate is typically given by the code of a corresponding validator such that the mathematical and the validator semantics should coincide. The following example illustrates the usage of an existing validation framework to provide an implementation of the predicates of a signature.

Example 10 (Implementation of predicates of a signature)**.** *Let us consider a system for international money transfers.* IBAN *(International Bank Account Number) is the standard for identifying bank accounts internationally. Some countries have not adopted this standard and, for money transfer to these countries, a special* clearing code *is needed in combination with the plain* account number. BIC *(Bank Identifier Code) is the standard for identifying banks globally.*

A form for international money transfers should contain (at least) the input fields bic, iban, account *and* clearingCode*. Supposing that the currency is Euro, this form should also contain the input fields* amountEuros *and* amountCents*. Moreover, this form should satisfy the following requirements:*

1. *The BIC code of the beneficiary's bank is required.*
2. *Either the IBAN or both clearing code and account number are required.*
3. *The amount to transfer must be between* 0.01 *and* 100000.00 *Euros.*

Table 2.3 shows a signature $\Phi = (\Pi^{\Phi}, \alpha^{\Phi})$ *which contains predicates used to specify data validation constraints.*

Table 2.3: A data validation signature Φ

$\pi \in \Pi^{\Phi}$	$\alpha^{\Phi}(\pi)$	Proposed vis.	Semantic interpretation
`[required]`	$1 \xrightarrow{a} 2$	X •—f→ Y	$\forall x \in X : f(x)$ defined
`[exactly-one-null]`	$1 \xrightarrow{a} 2$, $1 \xrightarrow{b} 3$	X —f→ Y, X —g→ Z, [eon]	$\forall x \in X : (f(x)$ defined and $g(x)$ undefined) or ($f(x)$ undefined and $g(x)$ defined)
`[all-or-none-null]`	$1 \xrightarrow{a} 2$, $1 \xrightarrow{b} 3$	X —f→ Y, X —g→ Z, [aonn]	$\forall x \in X : (f(x)$ defined and $g(x)$ defined) or ($f(x)$ undefined and $g(x)$ undefined)
`[cross-range-((`m_1, n_1`),(`m_2, n_2`))]`	$1 \xrightarrow{a} 2$, $1 \xrightarrow{b} 2$	X —f→ Int, X —g→ Int, [m₁.n₁−m₂.n₂]	$\forall x \in X : (m_1, n_1) \leq (f(x), g(x)) \leq (m_2, n_2)$
`[range(`m, n`)]`	$1 \xrightarrow{a} 2$	X —f→ Int, [m−n]	$\forall x \in X : m \leq f(x) \leq n$

Note that in the semantic interpretation of the `[cross-range]`*, the symbol* $\leq$ *refers to the lexicographical order.*

Figure 2.5 shows a specification $\mathfrak{P} = (P, C^{\mathfrak{P}} : \Phi)$ *representing an object-oriented structural model of the form above. The form is represented by the node* Payment *while the input fields are represented by the arrows* bic, iban, account, clearingCode, amountEuros *and* amountCents*.*

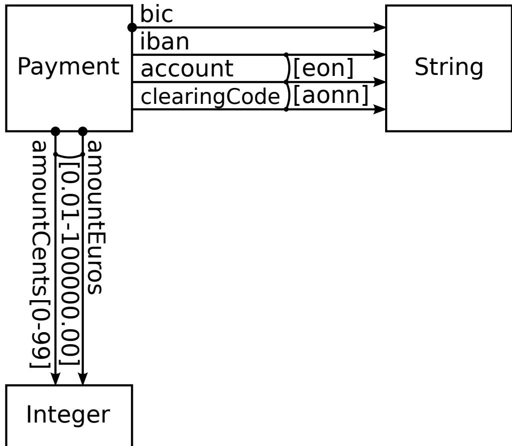


Figure 2.5: A specification $\mathfrak{P} = (P, C^{\mathfrak{P}} : \Phi)$

In $\mathfrak{P}$, the requirement 1 is enforced by the atomic constraint (`[required]`, δ_1) on the arrow bic*. This atomic constraint ensures that the user provides a value in the input field* bic*. Moreover, the requirement 2 is enforced in $\mathfrak{P}$ by two atomic constraints: (`[exactly-one-null]`, δ_2) on the arrows* iban *and* account *together with (`[all-or-none-null]`, δ_3) on the arrows* account *and* clearingCode*. These atomic constraints ensure that a user provides values in either the input field* iban *or both the input fields* account *and* clearingCode*. Furthermore, the requirement 3 is enforced in $\mathfrak{P}$ by the atomic constraint (`[cross-range`$((0,1),(100000,0))$`]`, δ_4) on the arrows* amountEuros *and* amountCents*. This atomic constraint ensures that the user provides values in the input fields* amountEuros *and* amountCents *which add up to a value within the range* 0.01 *to* 100000.00*. In addition, the atomic constraint (`[range`$(0,99)$`]`, δ_5) on the arrow* amountCents *ensures that a user provides a value in the input field* amountCents *within the range* 0 *to* 99*.*

For the signature Φ, it is possible to base the implementation of each predicate on the SHIP Validator [49, 61]. The XMI serialisation of the specification $\mathfrak{P} = (P, C^{\mathfrak{P}} : \Phi)$ (see Listing 2.1) can be transformed to a Java class tagged by Java annotations compatible with the SHIP Validator (see Listing 2.2). For each atomic constraint $(\pi, \delta) \in C^{\mathfrak{P}}$, a corresponding Java annotation is attached to the getter methods of the Java class. Note that an atomic constraint on a single arrow, e.g., (`[required]`, δ_1) on the arrow bic*, translates to a single Java annotation, e.g.,* `@Required` *on the method* `getBic()`*. Likewise, an atomic constraint on multiple arrows, e.g., (`[exactly-one-null]`, δ_2) on the arrows* iban *and* account*, translates to multiple Java annotations, e.g.,* `@ExactlyOneNull` *on the methods* `getIban()` *and* `getAccount()`*. The interested reader can download a proof-of-concept implementation of a code generator from [14].*

Listing 2.1: XMI serialisation of the specification $\mathfrak{P} = (P, C^{\mathfrak{P}}\colon\Phi)$

```
<?xml version="1.0" encoding="ASCII"?>
<no.hib.dpf.metamodel:Specification
xmlns:no.hib.dpf.metamodel="http://no.hib.dpf.metamodel"
id="9090a2ec-0e36-4fcc-8f04-3a0226f0a938" name="P">

 <node id="525d2a64-66e1-42f8-aec9-9f186379a77b" name="Payment"/>
 <node id="d3ae4964-d091-41d7-9127-09856b3ce316" name="String"/>
 <node id="0cac0671-a7e0-4d99-8216-14d24f186375" name="Integer"/>

 <arrow id="b5a45cda-3ee0-42a0-a568-81f9e92d7e25" name="bic" source="//@node.0" target="//@node.1"/>
 <arrow id="ad030229-b66c-40b5-8f7f-59f1a25e24a8" name="iban" source="//@node.0" target="//@node.1"/>
 <arrow id="1d54b8c6-a51b-4858-ade9-0a66522b80eb" name="account" source="//@node.0" target="//@node.1"/>
 <arrow id="2c4b8f89-dc27-44e6-bdb4-a0e298c26f85" name="clearingCode" source="//@node.0" target="//@node.1"/>
 <arrow id="07a4001b-4c8e-461f-a845-4ac985b0c36d" name="amountEuros" source="//@node.0" target="//@node.2"/>
 <arrow id="7559cb35-863a-49dd-a2b3-3e9e893c1356" name="amountCents" source="//@node.0" target="//@node.2"/>

 <constraints id="33003eb9-d287-4bd8-9a28-ccf6d3ea9ee0" type="[required]">
   <arrow source="//@arrow.0" />
 </constraints>

 <constraints id="33003eb6-7987-4558-ba28-aaf693349ee0" type="[not-required]">
   <arrow source="//@arrow.1" />
   <arrow source="//@arrow.2" />
   <arrow source="//@arrow.3" />
   <arrow source="//@arrow.4" />
   <arrow source="//@arrow.5" />
 </constraints>

 <constraints id="e0661dc3-0620-44e6-af54-07bf14875c16" type="[exactly-one-null]">
   <arrow source="//@arrow.1" />
   <arrow source="//@arrow.2" />
 </constraints>

 <constraints id="1160e483-b701-4c23-9641-7e73909de528" type="[all-or-none-null]">
   <arrow source="//@arrow.2" />
   <arrow source="//@arrow.3" />
 </constraints>

 <constraints id="e1f2bab1-b58c-4273-97bb-d0cdd14abe45" type="[cross-range]">
   <param name="m1" value="0" />
   <param name="n1" value="01" />
   <param name="m2" value="100000" />
   <param name="n2" value="00" />
   <arrow source="//@arrow.4" />
   <arrow source="//@arrow.5" />
 </constraints>

 <constraints id="9132c6e8-7af9-4fc6-8b67-afac0471b13b" type="[range]">
   <param name="min" value="0" />
   <param name="max" value="99" />
   <arrow source="//@arrow.5" />
 </constraints>

</no.hib.dpf.metamodel:Specification>
```

Listing 2.2: Java class generated by transformation

```
1  public class Payment {
2
3    private String bic;
4    private String iban;
5    private String account;
6    private String clearingCode;
7
8    private int amountEuros;
9    private int amountCents;
10
11   @Required
12   public String getBic() {
13     return bic;
14   }
15
16   @ExactlyOneNull
17   @NotRequired
18   public String getIban() {
19     return iban;
20   }
21
22   @ExactlyOneNull
23   @AllOrNoneNull
24   @NotRequired
25   public String getAccount() {
26     return account;
27   }
28
29   @AllOrNoneNull
30   @NotRequired
31   public String getClearingCode() {
32     return clearingCode;
33   }
34
35   @IntRange(min=0,max=100000)
36   @CrossRange
37   public int getAmountEuros(){
38     return this.amountEuros;
39   }
40
41   @IntRange(min=0,max=99)
42   @CrossRange
43   public int getAmountCents(){
44     return this.amountCents;
45   }
46
47 }
```

These Java annotations are in turn transformed into executable tests by the SHIP Validator. The interested reader can consult [49, 61] for details about the implementation and execution of these tests. Note that the idea of using annotations to hide the actual validation code and, at the same time, tag the properties to be tested, allow the constraints to be easily integrated into existing code. Besides, the validation aspects of the system remain well separated from the application aspects. This separation of concerns facilitates the transformation of the atomic constraints into actual working code.

2.3 Typing and conformance

In DPF, a specification $\mathfrak{S}$ is said to be typed by a graph T if there exists a graph homomorphism $\iota : S \to T$, called the *typing morphism*, between the underlying graph of the specification $\mathfrak{S}$ and the graph T. A specification $\mathfrak{S}$ is said to conform to a specification $\mathfrak{T}$ if there exists a typing morphism $\iota : S \to T$ between the underlying graphs of $\mathfrak{S}$ and $\mathfrak{T}$ such that (S, ι) is a valid instance of $\mathfrak{T}$; i.e., such that ι satisfies the atomic constraints $C^{\mathfrak{T}}$.

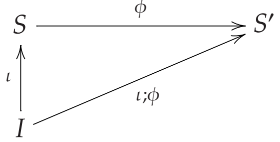
Definition 11 (Typed specification). *Given a signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$ and a graph T, a specification $\mathfrak{S} = (S, C^{\mathfrak{S}}:\Sigma)$ typed by T is a specification $\mathfrak{S}$ together with a graph homomorphism $\iota : S \to T$, called the* typing morphism.

Definition 12 (Conformant specification). *Given a signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$ and a specification $\mathfrak{T} = (T, C^{\mathfrak{T}}:\Sigma)$, a specification $\mathfrak{S} = (S, C^{\mathfrak{S}}:\Sigma)$ which conforms to $\mathfrak{T}$ is a specification $\mathfrak{S}$ together with a typing morphism $\iota : S \to T$ such that $(S, \iota) \in \mathbf{Inst}(\mathfrak{T})$.*

2.4 Specification morphism

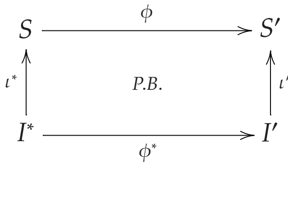
In DPF, the relation between specifications is represented by *specification morphisms*. Specification morphisms are graph homomorphisms between the underlying graphs of specifications. These graph homomorphisms induce a translation of instances of graphs.

Proposition 1 (Translation of instances of graphs). *Each graph homomorphism $\phi : S \to S'$ induces a functor $\phi_{\bullet} : \mathbf{Inst}(S) \to \mathbf{Inst}(S')$ with $\phi_{\bullet}(I, \iota) = (I, \iota;\phi)$ for all $(I, \iota) \in \mathbf{Inst}(S)$.*



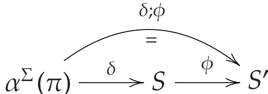
$$\mathbf{Inst}(S) \xrightarrow{\phi_{\bullet}} \mathbf{Inst}(S')$$

Moreover, each graph homomorphism $\phi : S \to S'$ induces a functor $\phi^{\bullet} : \mathbf{Inst}(S') \to \mathbf{Inst}(S)$ with $\phi^{\bullet}(I', \iota')$ given by the pullback $(I^{*}, \phi^{*} : I^{*} \to I', \iota^{*} : I^{*} \to S)$ of the span $S \xrightarrow{\phi} S' \xleftarrow{\iota'} I'$ [31].



In addition, these graph homomorphisms should preserve atomic constraints.

Definition 13 (Specification morphism)**.** *Given two specifications $\mathfrak{S} = (S, C^{\mathfrak{S}} \colon \Sigma)$ and $\mathfrak{S}' = (S', C^{\mathfrak{S}'} \colon \Sigma)$, a specification morphism $\phi : \mathfrak{S} \to \mathfrak{S}'$ is a graph homomorphism $\phi : S \to S'$ such that $(\pi, \delta) \in C^{\mathfrak{S}}$ implies $(\pi, \delta; \phi) \in C^{\mathfrak{S}'}$.*



Remark 3 (Subspecification)**.** *A specification $\mathfrak{S} = (S, C^{\mathfrak{S}} \colon \Sigma)$ is a subspecification of a specification $\mathfrak{S}' = (S', C^{\mathfrak{S}'} \colon \Sigma)$, written $\mathfrak{S} \sqsubseteq \mathfrak{S}'$, if and only if S is a subgraph of S' and the inclusion graph homomorphism $inc : S \hookrightarrow S'$ defines a specification morphism $inc : \mathfrak{S} \hookrightarrow \mathfrak{S}'$.*

Remark 4 (Graph homomorphism and atomic constraints)**.** *Any graph homomorphism $\phi : S \to S'$ induces a translation of atomic constraints; i.e., for any specification $\mathfrak{S} = (S, C^{\mathfrak{S}} \colon \Sigma)$ we obtain a specification $\phi(\mathfrak{S}) = (S', C^{\phi(S)} \colon \Sigma)$ with $C^{\phi(S)} = \phi(C^{\mathfrak{S}}) = \{(\pi, \delta; \phi) \mid (\pi, \delta) \in C^{\mathfrak{S}}\}$.*

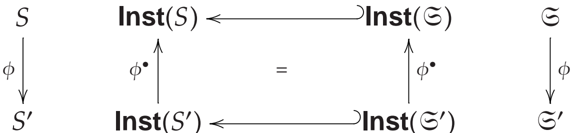
Based on this remark, the condition for specification morphisms can be reformulated as follows: a specification morphism $\phi : \mathfrak{S} \to \mathfrak{S}'$ is a graph homomorphism $\phi : S \to S'$ such that $\phi(\mathfrak{S}) \sqsubseteq \mathfrak{S}'$, i.e., $C^{\phi(S)} = \phi(C^{\mathfrak{S}}) \subseteq C^{\mathfrak{S}'}$.

Given a signature Σ, the category of specifications is defined as follows:

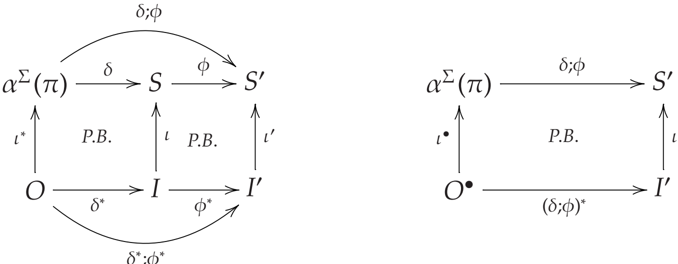
Definition 14 (Category of specifications)**.** *Given a signature $\Sigma = (\Pi^\Sigma, \alpha^\Sigma)$, the category* **Spec**$(\Sigma)$ *has all specifications $\mathfrak{S} = (S, C^{\mathfrak{S}} \colon \Sigma)$ as objects and all specification morphisms $\phi : \mathfrak{S} \to \mathfrak{S}'$ as morphisms between specifications $\mathfrak{S}$ and $\mathfrak{S}'$.*

The associativity of composition of graph homomorphism ensures that the composition of two specification morphisms is a specification morphism as well and that the composition of specification morphisms is associative. Moreover, the identity graph homomorphisms $id^S : S \to S$ define identity specification morphisms $id^{\mathfrak{S}} : \mathfrak{S} \to \mathfrak{S}$ and ensure that identity specification morphisms are left and right neutral with respect to composition.

Proposition 2 (Specification morphisms and category of instances)**.** *For any specification morphism $\phi : \mathfrak{S} \to \mathfrak{S}'$, we have $\phi^\bullet(\mathbf{Inst}(\mathfrak{S}')) \subseteq \mathbf{Inst}(\mathfrak{S})$; i.e., the functor $\phi^\bullet : \mathbf{Inst}(S') \to \mathbf{Inst}(S)$ restricts to a functor $\phi^\bullet : \mathbf{Inst}(\mathfrak{S}') \to \mathbf{Inst}(\mathfrak{S})$.*



Proof. The proof is given by the result that the composition of two pullbacks is again a pullback [12] and by the assumption that $[\![\pi]\!]^\Sigma$ is closed under isomorphisms (see Definition 8), as shown in [31].



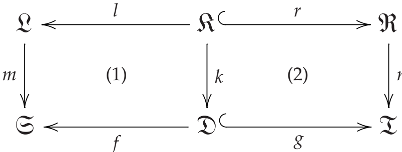
□

2.5 Specification transformation

In this thesis, *specification transformation* is based on *transformation rules* [36, 47]. A transformation rule $t = \mathfrak{L} \xleftarrow{l} \mathfrak{K} \xhookrightarrow{r} \mathfrak{R}$ consists of three specifications $\mathfrak{L}$, $\mathfrak{K}$ and $\mathfrak{R}$. $\mathfrak{L}$ and $\mathfrak{R}$ are the *left-hand side* and *right-hand side* of the transformation rule, respectively, while $\mathfrak{K}$ is their interface. $\mathfrak{L} \setminus l(\mathfrak{K})$ describes the part of a specification which is to be deleted, $\mathfrak{R} \setminus \mathfrak{K}$ describes the part to be added, and $\mathfrak{K}$ describes the part which has to exist to apply the rule, in which only renaming modifications are possible. Note that the specification morphism $l : \mathfrak{K} \to \mathfrak{L}$ is injective in order to allow for renaming. An *application of transformation rule* means finding a match for the left-hand side $\mathfrak{L}$ in a source specification $\mathfrak{S}$ and replacing $\mathfrak{L}$ with $\mathfrak{R}$, leading to a target specification $\mathfrak{T}$.

Definition 15 (Transformation rule)**.** *A transformation rule*
$t = \mathfrak{L} \xleftarrow{l} \mathfrak{K} \xhookrightarrow{r} \mathfrak{R}$ *consists of specifications $\mathfrak{L}$, $\mathfrak{K}$ and $\mathfrak{R}$, called left-hand side, interface and right-hand side, respectively, an injective specification morphism $l : \mathfrak{K} \to \mathfrak{L}$ and an inclusion specification morphism $r : \mathfrak{K} \hookrightarrow \mathfrak{R}$.*

Definition 16 (Application of transformation rule)**.** *Given a transformation rule* $t = \mathfrak{L} \xleftarrow{l} \mathfrak{K} \xhookrightarrow{r} \mathfrak{R}$ *, a specification* $\mathfrak{S}$ *and an injective specification morphism* $m : \mathfrak{L} \to \mathfrak{S}$*, called the match, an application of transformation rule* $\mathfrak{S} \xRightarrow{<t,m>} \mathfrak{T}$ *from a specification* $\mathfrak{S}$ *to a specification* $\mathfrak{T}$ *is given by the following double-pushout (DPO) [36], where (1) and (2) are pushouts in the category* **Spec**(Σ) *(see Propositions 6 and 7):*



Definition 17 (Specification transformation)**.** *A specification transformation* $\mathfrak{S} \xRightarrow{*} \mathfrak{S}'$ *consists of a sequence of applications of transformation rules on* $\mathfrak{S}$*.*

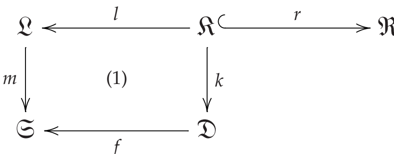
When an application of a transformation rule t via a match m is performed, all nodes, arrows and atomic constraints which are in the image of m but not in the image of $l; m$ are deleted from the specification $\mathfrak{S}$. In general, the deleted part does not need to be a valid specification, but the remaining specification $\mathfrak{D} := (\mathfrak{S} \setminus m(\mathfrak{L})) \cup m(\mathfrak{K})$ still has to be a valid specification with no dangling arrows or dangling atomic constraints. This means that the match m has to satisfy a *gluing condition* [36], which ensures that the gluing of $\mathfrak{L} \setminus \mathfrak{K}$ and $\mathfrak{D}$ is equal to $\mathfrak{S}$.

Definition 18 (Gluing condition)**.** *Given a transformation rule* $t = \mathfrak{L} \xleftarrow{l} \mathfrak{K} \xhookrightarrow{r} \mathfrak{R}$ *, a specification* $\mathfrak{S}$ *and a match* $m : \mathfrak{L} \to \mathfrak{S}$*:*

- *The* gluing points *GP consist of the nodes and arrows in L which are not deleted by t, i.e.,* $GP = l_N(K_N) \cup l_A(K_A) = l(K)$*.*
- *The* dangling arrow points *DAP consist of the nodes in L whose images under m are the source or target of an arrow in S which does not belong to m(L), i.e.,* $DAP = \{X \in L_N \mid \exists f \in S_A \setminus m_A(L_A) : src(f) = m_N(X)$ *or* $trg(f) = m_N(X)\}$*.*
- *The* dangling atomic constraint points *DACP consist of the nodes and arrows in L whose images under m are in the image of the graph homomorphism* δ *of an atomic constraint* (π, δ) *in* $C^{\mathfrak{S}}$ *which does not belong to m(L), i.e.,* $DACP = \{X \in L_N \mid \exists(\pi, \delta) \in C^{\mathfrak{S}} \setminus m(C^{\mathfrak{L}}) : m_N(X) \in \delta_N(\alpha(\pi))\} \cup \{f \in L_A \mid \exists(\pi, \delta) \in C^{\mathfrak{S}} \setminus m(C^{\mathfrak{L}}) : m_A(f) \in \delta_A(\alpha(\pi))\}$*.*

The transformation rule t and the match m satisfy the gluing condition if all identification points and all dangling points are also gluing points, i.e., $DAP \cup DACP \subseteq GP$*.*

Definition 19 (Applicability of transformation rules)**.** *A transformation rule* $t = \mathfrak{L} \xleftarrow{l} \mathfrak{K} \xhookrightarrow{r} \mathfrak{R}$ *is applicable to a specification* $\mathfrak{S}$ *via a match* $m : \mathfrak{L} \to \mathfrak{S}$ *if there exists a context specification* $\mathfrak{D}$ *such that (1) is a pushout in the category* **Spec**(Σ).



Remark 5 (Existence and uniqueness of context specification)**.** *Given a transformation rule* $t = \mathfrak{L} \xleftarrow{l} \mathfrak{K} \xhookrightarrow{r} \mathfrak{R}$*, a specification* $\mathfrak{S}$ *and a match* $m : \mathfrak{L} \to \mathfrak{S}$*, the context specification* $\mathfrak{D}$ *together with the pushout (1) exist if and only if the gluing condition is satisfied. If* $\mathfrak{D}$ *exists, it is unique up to isomorphism.*

The proof of the existence and uniqueness of context specification can be provided by extending the results in [36] from graph transformation to DPF.

A specification transformation may show two kinds of non-determinism [36]. Firstly, there may be more than one applicable transformation rule. Secondly, there may be more than one match for a transformation rule in the source specification. In both cases, the choice may be arbitrary. Some degree of determinism may be achieved by controlling the flow of the application of transformation rules.

In addition to these two kinds of non-determinism, a specification transformation is, in general, non-terminating [36]. A specification transformation $\mathfrak{S} \xRightarrow{*} \mathfrak{T}$ is *terminating* if no more transformation rules can be applied to $\mathfrak{T}$. However, given a set of transformation rules, there may be two specification transformations $\mathfrak{S} \xRightarrow{*} \mathfrak{T}'$ and $\mathfrak{S} \xRightarrow{*} \mathfrak{T}''$ leading to two non-isomorphic target specifications $\mathfrak{T}'$ and $\mathfrak{T}''$. A set of transformation rules is *confluent* if, for each pair of specification transformations $\mathfrak{S} \xRightarrow{*} \mathfrak{T}'$ and $\mathfrak{S} \xRightarrow{*} \mathfrak{T}''$, there exists a specification $\mathfrak{X}$ together with specification transformations $\mathfrak{T}' \xRightarrow{*} \mathfrak{X}$ and $\mathfrak{T}'' \xRightarrow{*} \mathfrak{X}$.

A specification transformation which is terminating and confluent is said to have *functional behaviour* [36]. The formalisation of termination and confluence in view of DPF is outside the scope of this thesis and will be investigated in future work (see Section 2.8)

Among techniques for controlling the application of transformation rules and achieving functional behaviour are the *negative application conditions* (NACs) [36, 80]. NACs are used to forbid applications of a transformation rule. Since non-deleting transformation rules (i.e., transformation rules which do not delete any specification element) can be applied multiple times via the same match, it is necessary to require that the right-hand side of each transformation rule always defines a NAC for the transformation rule itself. This is to ensure that a transformation rule is applied only once via a given match.

Another technique for controlling the application of transformation rules is the *layering of transformation rules* [36, 80]. In this technique, each transformation rule is assigned to a numbered layer based on its order of application. The transformation rules at each layer are applied before the transformation rules at the next layer.

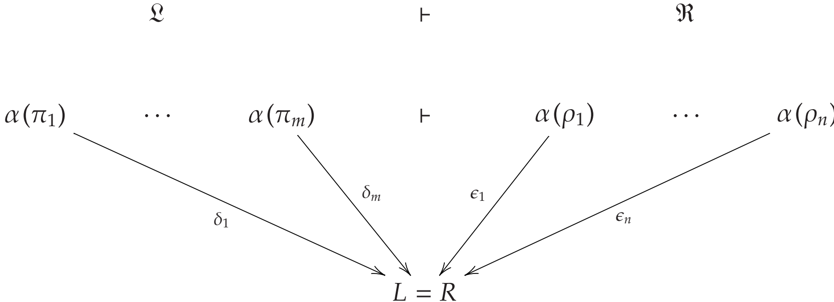
2.6 Specification entailment

Recall that a specification consists of an underlying graph together with a set of atomic constraints which are specified by means of predicates of a signature. Due to Definition 9, for any specification $\mathfrak{S} = (S, C^{\mathfrak{S}}: \Sigma)$, the atomic constraints $\{(\pi_1, \delta_1), \ldots, (\pi_n, \delta_n)\} = C^{\mathfrak{S}}$ are implicitly conjunctively connected.

In addition to this implicit conjunction, it would be desirable to define other relations between atomic constraints. Defining these relations can be regarded as describing properties of the semantic interpretation of predicates of a signature. For example, according to the semantic interpretation $[\![\texttt{[mult(}m,n\texttt{)]}]\!]^{\Sigma}$ of the signature Σ (see Table 2.1), a valid instance of the atomic constraint $(\texttt{[mult(}2,3\texttt{)]}, \delta)$ is also a valid instance of (or satisfies) the atomic constraint $(\texttt{[mult(}1,4\texttt{)]}, \delta)$. This kind of relation is called predicate dependency in [31, 89]. However, defining relations between single atomic constraints may not be sufficient.

In this thesis, *specification entailments* are used to express relations between conjunctively connected sets of atomic constraints. A specification entailment has the structure $Left \vdash Right$, where both premise ($Left$) and conclusion ($Right$) are specifications with the same underlying graph.

Definition 20 (Specification entailment)**.** *A specification entailment $e = \mathfrak{L} \vdash \mathfrak{R}$ consists of two specifications $\mathfrak{L} = (L, C^{\mathfrak{L}}: \Sigma)$ and $\mathfrak{R} = (R, C^{\mathfrak{R}}: \Sigma)$, called the premise and the conclusion, respectively, with the same underlying graph $L = R$, called the context graph.*



A specification entailment is valid if and only if all instances of the premise are also instances of the conclusion.

Definition 21 (Semantic interpretation and specification entailment)**.** *A specification entailment $e = \mathfrak{L} \vdash \mathfrak{R}$, with $\mathfrak{L} = (L, C^{\mathfrak{L}} \colon \Sigma)$ and $\mathfrak{R} = (R, C^{\mathfrak{R}} \colon \Sigma)$, is valid for a semantic interpretation $[\![..]\!]^{\Sigma}$ of a signature Σ if and only if* $\textbf{Inst}(\mathfrak{L}) \subseteq \textbf{Inst}(\mathfrak{R})$.

The following example illustrates the usage of specification entailments to express relations between multiplicity and surjectivity constraints.

Example 11 (Specification entailment)**.** *Building upon Example 8, Figure 2.6 shows a specification entailment $e = \mathfrak{L} \vdash \mathfrak{R}$ with:*

$$\mathfrak{L} = (L, C^{\mathfrak{L}} = \{([\texttt{mult}(0,n)], \delta_1), ([\texttt{inverse}], \delta_2), ([\texttt{surjective}], \delta_3)\} : \Sigma)$$
$$\mathfrak{R} = (R, C^{\mathfrak{R}} = \{([\texttt{mult}(1,n)], \epsilon_1), ([\texttt{inverse}], \epsilon_2), [\texttt{surjective}], \epsilon_3)\} : \Sigma)$$

$$L = R = X \underset{g}{\overset{f}{\rightleftarrows}} Y$$

One can show that according to the semantic interpretation $[\![..]\!]^{\Sigma}$ of the signature Σ (see Table 2.1) the requirement $\textbf{Inst}(\mathfrak{L}) \subseteq \textbf{Inst}(\mathfrak{R})$ is satisfied:

$$\begin{array}{rcl} \textit{Since } g \textit{ is surjective:} & & \forall x \in X\ \exists y \in Y : x \in g(y) \\ f, g \textit{ inverse gives} & \Rightarrow & \forall x \in X\ \exists y \in Y : y \in f(x) \\ f \textit{ total gives} & \Rightarrow & \forall x \in X : |f(x)| \geq 1 \end{array}$$

Note that for the specification entailment above, it is trivial to prove that also $\textbf{Inst}(\mathfrak{L}) \supseteq \textbf{Inst}(\mathfrak{R})$, concluding that $\textbf{Inst}(\mathfrak{L}) \equiv \textbf{Inst}(\mathfrak{R})$.

As mentioned, each specification entailment is defined over a given context graph. From these specification entailments, one may induce transformation rules, which can be applied to existing specifications.

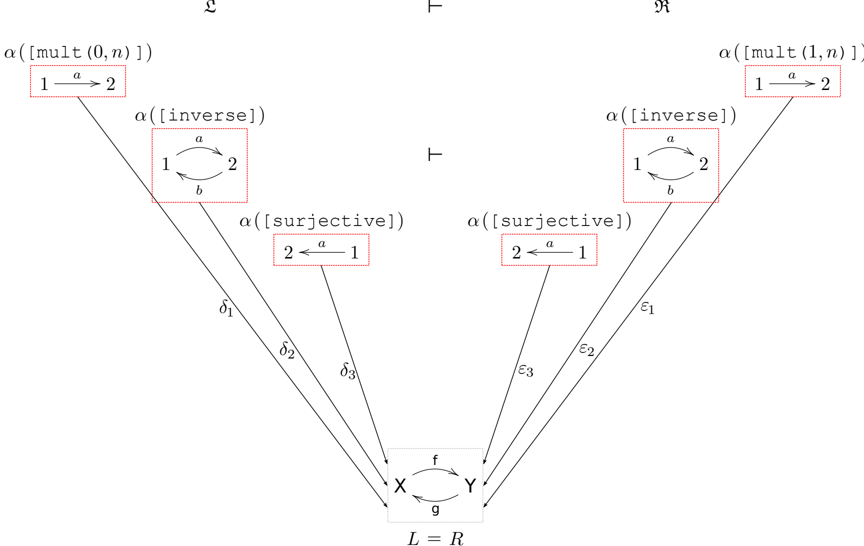


Figure 2.6: A specification entailment $e = \mathfrak{L} \vdash \mathfrak{R}$

Proposition 3 (Specification entailment and transformation rule). *Each specification entailment $e = \mathfrak{L} \vdash \mathfrak{R}$, with $\mathfrak{L} = (L, C^{\mathfrak{L}} : \Sigma)$ and $\mathfrak{R} = (R, C^{\mathfrak{R}} : \Sigma)$, induces a transformation rule $t = \mathfrak{L} \xleftarrow{l} \mathfrak{K} \xhookrightarrow{r} \mathfrak{R}$, with $\mathfrak{K} = (K, C^{\mathfrak{K}} : \Sigma)$, where $K = L = R$ and $C^{\mathfrak{K}} = C^{\mathfrak{L}} \cap C^{\mathfrak{R}}$.*

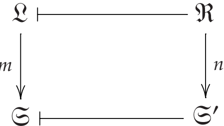
The following example illustrates the transformation rule which is induced by the specification entailment for multiplicity and surjectivity constraints.

Example 12 (Specification entailment and transformation rule). *Building upon Example 11, Table. 2.4 shows the transformation rule $t = \mathfrak{L} \xleftarrow{l} \mathfrak{K} \xhookrightarrow{r} \mathfrak{R}$ induced by the specification entailment $e = \mathfrak{L} \vdash \mathfrak{R}$.*

Table 2.4: The transformation rule $t = \mathfrak{L} \leftarrow \mathfrak{K} \hookrightarrow \mathfrak{R}$ induced by the specification entailment $e = \mathfrak{L} \vdash \mathfrak{R}$

Rule	$\mathfrak{L}$	$\mathfrak{K}$	$\mathfrak{R}$
t	X f [0..n] [inv] [surj] g Y	X f [inv] [surj] g Y	X f [1..n] [inv] [surj] g Y

Proposition 4 (Embedding of specification entailment)**.** *Given a transformation rule $t = \mathfrak{L} \xleftarrow{l} \mathfrak{K} \xhookrightarrow{r} \mathfrak{R}$ induced by a specification entailment $e = \mathfrak{L} \vdash \mathfrak{R}$, and a specification $\mathfrak{S} = (S, C^{\mathfrak{S}}{:}\Sigma)$ together with a match $m : \mathfrak{L} \rightarrow \mathfrak{S}$, each application of transformation rule $\mathfrak{S} \xRightarrow{<t,m>} \mathfrak{S}'$ induces a specification entailment $\mathfrak{S} \vdash \mathfrak{S}'$ with $\mathfrak{S}' = (S', C^{\mathfrak{S}'}{:}\Sigma)$.*



Given a semantic interpretation $[\![..]\!]^{\Sigma}$ of a signature Σ, an induced specification entailment $\mathfrak{S} \vdash \mathfrak{S}'$ is valid as long as $\mathfrak{L} \vdash \mathfrak{R}$ is valid.

Proof. The proof is given by showing that if $\textbf{Inst}(\mathfrak{L}) \subseteq \textbf{Inst}(\mathfrak{R})$ then $\textbf{Inst}(\mathfrak{S}) \subseteq \textbf{Inst}(\mathfrak{S}')$.

Firstly, we have to show that $(I^S, \iota^S) \vDash m(C^{\mathfrak{L}})$ implies that $(I^S, \iota^S) \vDash m(C^{\mathfrak{R}})$.

Suppose that $(I^S, \iota^S) \vDash m(C^{\mathfrak{L}})$. By Proposition 2 this holds if and only if $m^{\bullet}(I^S, \iota^S) \vDash C^{\mathfrak{L}}$. The specification entailment $\mathfrak{L} \vdash \mathfrak{R}$ implies that $m^{\bullet}(I^S, \iota^S) \vDash C^{\mathfrak{R}}$. Proposition 2 implies that $(I^S, \iota^S) \vDash m(C^{\mathfrak{R}})$.

Secondly, we have to show that $(I^S, \iota^S) \vDash m(C^{\mathfrak{R}})$ implies that $(I^S, \iota^S) \vDash m(C^{\mathfrak{S}'})$.

Suppose that $(I^S, \iota^S) \vDash C^{\mathfrak{S}}$; i.e., $(I^S, \iota^S) \vDash m(C^{\mathfrak{L}})$ and $(I^S, \iota^S) \vDash C^{\mathfrak{S}} \setminus m(C^{\mathfrak{L}})$. This implies that $(I^S, \iota^S) \vDash m(C^{\mathfrak{R}})$ and $(I^S, \iota^S) \vDash C^{\mathfrak{S}} \setminus m(C^{\mathfrak{L}})$. Proposition 8 implies that $C^{\mathfrak{S}'} = C^{\mathfrak{S}} \setminus m(C^{\mathfrak{L}}) \cup m(C^{\mathfrak{R}})$; i.e., $(I^S, \iota^S) \vDash m(C^{\mathfrak{S}'})$. □

The following example illustrates the embedding of the specification entailment for multiplicity and surjectivity constraints.

Example 13 (Embedding). *Building upon Examples 8 and 11, Figure 2.7 shows the specification entailment $\mathfrak{S} \vdash \mathfrak{S}'$ induced by the application of the transformation rule* $\mathfrak{S} \xRightarrow{<t,m>} \mathfrak{S}'$.

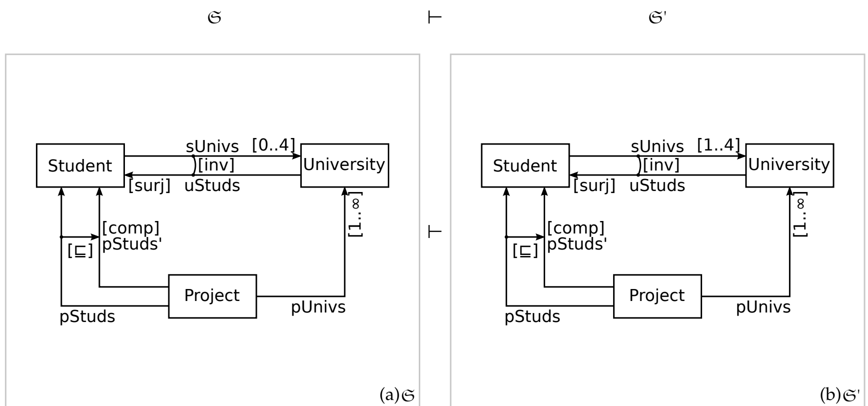


Figure 2.7: An embedding $\mathfrak{S} \vdash \mathfrak{S}'$ of the specification entailment $\mathfrak{L} \vdash \mathfrak{R}$

According to the semantic interpretation $[\![\texttt{[mult}(m,n)\texttt{]}]\!]^{\Sigma}$ *of the signature* Σ *(see Table 2.1), the set of valid instances of the atomic constraint* $(\texttt{[mult}(0,4)\texttt{]}, \delta_1) \in C^{\mathfrak{S}}$ *is larger than the set of valid instances of the atomic constraint* $(\texttt{[mult}(1,4)\texttt{]}, \delta_1) \in C^{\mathfrak{S}'}$. *However, because of the implicit conjunction of the atomic constraints, the set of valid instances of* $C^{\mathfrak{S}}$ *and* $C^{\mathfrak{S}'}$ *are equal.*

2.7 Related work

The formalisation of diagrammatic modelling has been extensively discussed in the literature.

E-graphs

The work in [35, 36] uses E-graphs to represent models and metamodels. An E-graph is a generalisation of an attributed graph [34] and consists of two sets of graph and data nodes, respectively, and three sets of graph arrows, node attribute arrows and arrow attribute arrows, respectively. The assignment of attributes to nodes is done by adding node attribute arrows from the graph nodes to the data nodes. The assignment of attributes to arrows is done by adding arrow attribute arrows from the graph arrows to the data nodes. Attributes of nodes and arrows are used to describe properties of nodes and arrows, which is similar to how attributes of classes in the UML metamodel are used to describe properties of model elements. Attributes of nodes can be represented in DPF by arrows from these nodes to nodes representing data types. The adoption of E-graphs rather than directed multi-graphs may represent a natural next step in the development of DPF.

Graph constraints

The work in [36] also uses graph constraints to express properties for graphs. A graph constraint is of the form $a : Premise \to Conclusion$ where a is a graph homomorphism. In case a is surjective, a graph constraint can be seen to correspond to a first-order implication of the form $\forall \overline{x} : P(\overline{x}) \to Q(\overline{x})$ where $\overline{x}$ denotes a list of variables. In case a is not surjective, however, a graph constraint corresponds to a first-order implication of the form $\forall \overline{x} : P(\overline{x}) \to (\exists \overline{y} : Q(\overline{x}, \overline{y}))$. For some of our predicates the semantics can be described by these first-order implications. This is the case, for example, for the predicates `[surjective]` and `[inverse]`. In such a way, some atomic constraints at metamodel level give rise to graph constraints at model level. For example, in Figure 2.4(a) the atomic constraint (`[surjective]`, δ_3) on the arrow uStuds represents a graph constraint c_1 in Table 2.5, while the atomic constraint (`[inverse]`, δ_2) on the arrows sUnivs and uStuds represents the graph constraints c_2 and c_3 in Table 2.5.[1]

Algebraic specifications

The work in [18] proposes an algebraic semantics for MOF to formalise the concepts of models, metamodels and conformance between them. Models are represented by terms while metamodels are represented by specifications in membership equational logic (MEL). This formal semantics is made executable by using Maude [24], which directly supports MEL specifications.

[1]The graph constraints in Table 2.5 come with negative constraints to avoid duplication of the conclusion; this detail is omitted.

Table 2.5: Graph constraints represented by the atomic constraints ([surjective], δ_3) and ([inverse], δ_2)

Premise	*Conclusion*
c_1: *uStuds* is surjective	
s:Student	s:Student ← :uStuds — u:University
c_2: *uStuds* is inverse of *sUnivs* and	
s:Student — :sUnivs → u:University	s:Student — :sUnivs → u:University; s:Student ← :uStuds — u:University
c_3: *sUnivs* is inverse of *uStuds*	
s:Student ← :uStuds — u:University	s:Student — :sUnivs → u:University; s:Student ← :uStuds — u:University

Constructive type theory

The work in [76] exploits the higher-order nature of constructive type theory to uniformly treat the syntax of models, metamodels, as well as MOF itself. Models are represented by terms (token models) and can also be represented by types (type models) by means of a reflection mechanism. This formal semantics ensures that correct typing corresponds to provably correct models and metamodels.

Epsilon

Epsilon (Extensible Platform of Integrated Languages for mOdel maNagement) [54] is a family of consistent and interoperable task-specific programming languages which can be used to interact with EMF models. The core of Epsilon is the Epsilon Object Language (EOL), an imperative language that combines the procedural style of JavaScript with the querying capabilities of OCL. In addition, Epsilon provides several task-specific languages for performing code generation, model transformation, model validation, etc. One of these task-specific languages is the Epsilon Validation Language (EVL). EVL extends OCL conceptually (as opposed to technically) to provide a number of features such as support for constraint dependency management and access to multiple models conforming to different metamodels.

Alloy

Alloy [2] is a modelling language which is capable of expressing complex structural and behavioural constraints. Model analysis in Alloy is based on the usage of first order logic to translate specifications into boolean expressions which are automatically evaluated by a boolean satisfiability problem (SAT) solver. Given a logical formula, Alloy attempts to find a model which satisfies the formula. Alloy models are checked by using the Alloy analyser which attempts to find counterexamples, within a limited scope, that violate the constraints of the system. Even though Alloy cannot prove the system's consistency in an infinite scope, the user receives immediate feedback about the system's consistency.

2.8 Conclusion and future work

In this chapter, we outlined DPF along with a formalisation of some of the fundamental concepts in MDE. DPF is an adaptation of the categorical sketch formalism, where the constraining constructs of modelling languages are represented by user-defined signatures in a more intuitive and adequate way. In particular, DPF is an extension of the Generalised Sketches formalism and aims to combine mathematical rigour with diagrammatic modelling.

This chapter is an adaptation of the formalisation of modelling and model transformation published in [79, 80, 82, 84]. Compared to the previous work, the specification transformation is extended to support deleting transformation rules. Moreover, the embedding of specification entailments is also revised to adopt deleting transformation rules.

Specification transformations constitute the basis for several techniques presented in this thesis. In future work, we will analyse termination and confluence in view of DPF.

Specification entailments are used to characterise relations between sets of conjunctive connected atomic constraints. In future work, we will investigate a deduction calculus which would give rise to more complex deductions such as new specification entailments from given ones.

A prototype tool for DPF [13] is available at [15]. The tool is implemented in Java as an Eclipse plug-in and relies on EMF and the Graphical Editing Framework (GEF) [44]. In future work, we will perform empirical studies to determine whether the benefits of DPF and its formal approach to MDE are observable.

CHAPTER 3

Constraint-Aware Model Versioning

In the previous chapter, we outlined DPF along with a formalisation of some of the fundamental concepts in MDE. In this chapter, we describe a formal approach to constraint-aware model versioning based on DPF; i.e., a formal approach to model versioning which handles constraints in model merging, conflict detection and conflict resolution.

3.1 Introduction

Evolution of models

In MDE, models are first-class entities of software development and undergo a complex evolution during their life-cycles. As a consequence, there is a growing need for techniques and tools to support model management activities such as version control.

Optimistic version control

In *optimistic* version control, each developer has a local (or working) copy of a software artefact. These local copies are modified independently and in parallel and, from time to time, local modifications are merged together. In the *centralised*[1] approach to optimistic version control, local modifications of each developer are merged into a central repository. In the *distributed* approach, in contrast, local modifications of each developer are merged into other developers' local copies. In both cases, the merge is performed using a *three-way* merging technique [62], which attempts to merge two versions of a software artefact relying on the common ancestor version from which both versions originated. This technique facilitates

[1] Also referred to as *copy-modify-merge* [75, 78].

conflict detection. Roughly speaking, conflicts may arise when the modifications are contradictory. They are resolved either manually or, when applicable, automatically.

Text-based VCS

Mainstream version control systems (VCSs), e.g., Subversion [4] and Git [40], target text-based artefacts. Hence, underlying techniques such as merging, conflict detection and conflict resolution are based on a per-line textual comparison [50]. Since the underlying structure of models is graph-based rather than text-based, these techniques are not suitable for MDE.

Graph-based VCS

To cope with this problem, a few prototype VCSs have been developed that target graph-based structures, e.g., [19, 64]. However, a uniform formalisation of model merging, conflict detection and conflict resolution in MDE is still debated in the literature. Research has lead to a number of findings in this field [62]. The interested reader may consult [22, 26, 78, 87, 88] for different approaches to model merging, conflict detection and conflict resolution. Unfortunately, these techniques consider only model elements and their conformance to the corresponding modelling language, e.g., well-formedness constraints. However, these techniques should also consider constraints added to model elements, e.g., multiplicity constraints. An interesting challenge is then to extend the current techniques by enabling version control of constraints.

Constraint-aware model versioning

In this chapter, we describe a formal approach to constraint-aware model versioning based on DPF; i.e., a formal approach to model versioning which handles constraints in model merging, conflict detection and conflict resolution.

The remainder of the chapter is structured as follows. Section 3.2 introduces model versioning through a running example. Section 3.3 discusses the calculation and representation of differences in view of DPF. Section 3.4 presents a synchronisation procedure which includes conflict detection and conflict resolution. In Section 3.5, the current research in model versioning is summarised. In Section 3.6, some concluding remarks and ideas for future work are presented.

3.2 Model versioning

The following example illustrates a usual scenario of concurrent development in MDE. Note that the example is intentionally kept simple, retaining only the details which are relevant for the discussion. The following notation is employed:

- *Specification* $\mathfrak{V}_i$: a version of a specification in the repository, e.g., $\mathfrak{V}_2$
- *Local copy* $\mathfrak{U}_i$, with $\mathfrak{U}$ for user: a local copy of the specification $\mathfrak{V}_i$, e.g., $\mathfrak{A}_2$, with $\mathfrak{A}$ for Alice.

Example 14 (Model versioning and conflict detection scenario)**.** *Let us consider an information system for the management of students, universities and projects. Suppose that two software developers, Alice and Bob, adopt an optimistic, centralised VCS. Figure 3.1 illustrates the interaction between Alice, Bob and the repository. Figure 3.2 shows the different versions of the specification being developed, while Table 3.1 shows the signature used to specify the atomic constraints in these specifications.*

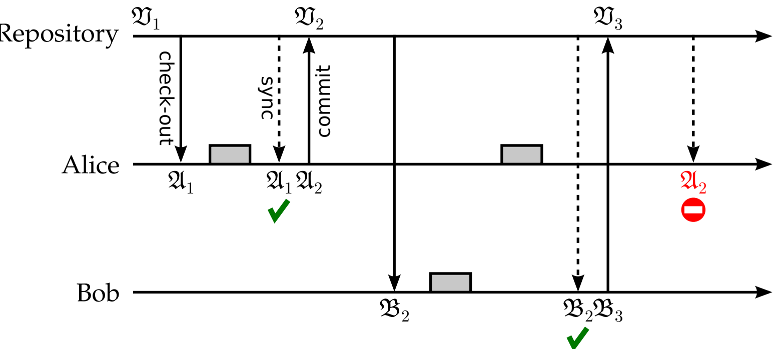


Figure 3.1: The timeline of the version control scenario

Table 3.1: The signature Σ

$\pi \in \Pi^{\Sigma}$	$\alpha^{\Sigma}(\pi)$	**Proposed vis.**	**Semantic interpretation**
`[mult(`m,n`)]`	$1 \xrightarrow{a} 2$	X $\xrightarrow{f}$ Y [m..n]	$\forall x \in X : m \leq \lvert f(x) \rvert \leq n$, with $0 \leq m \leq n$ and $n \geq 1$
`[injective]`	$1 \xrightarrow{a} 2$	X $\xrightarrow{f}$ Y [inj]	$\forall x, x' \in X : f(x) = f(x')$ implies $x = x'$
`[jointly-injective]`	$1 \xrightarrow{a} 2$, $1 \xrightarrow{b} 3$	X $\xrightarrow{f}$ Y, X $\xrightarrow{g}$ Z [ji]	$\forall x, x' \in X : f(x) = f(x')$ and $g(x) = g(x')$ implies $x = x'$
`[surjective]`	$1 \xrightarrow{a} 2$	X $\xrightarrow{f}$ Y [surj]	$\forall y \in Y\ \exists x \in X : y \in f(x)$
`[inverse]`	$1 \xrightarrow{a} 2$, $2 \xrightarrow{b} 1$	X $\xrightarrow{f}$ Y, Y $\xrightarrow{g}$ X [inv]	$\forall x \in X, \forall y \in Y : y \in f(x)$ if and only if $x \in g(y)$

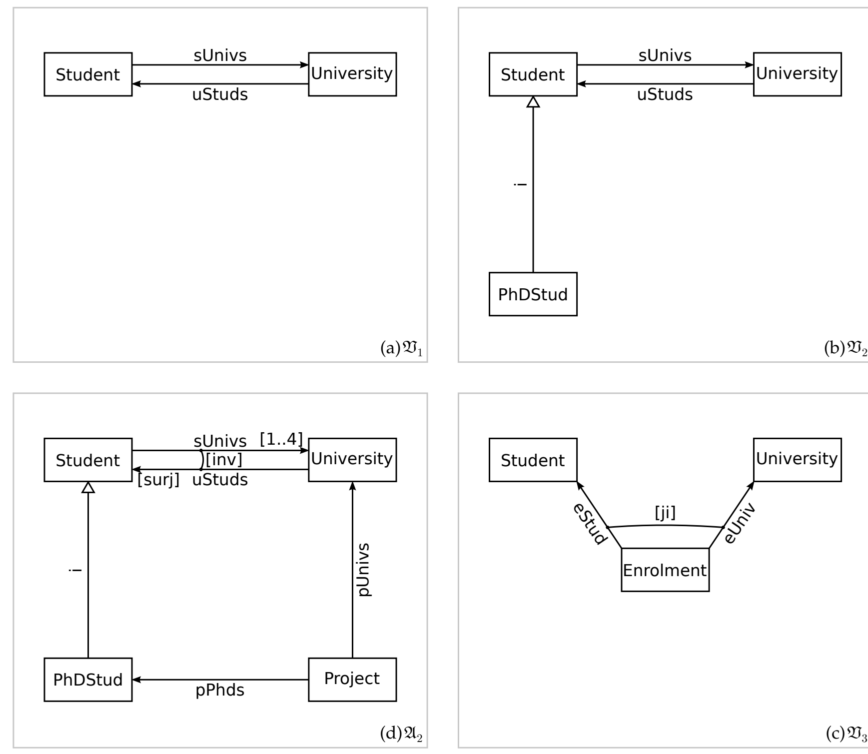


Figure 3.2: The specifications $\mathfrak{B}_1$, $\mathfrak{B}_2$, $\mathfrak{B}_3$ and $\mathfrak{A}_2$

Alice creates a local copy $\mathfrak{A}_1$ of the specification $\mathfrak{B}_1$ in the repository (see Figure 3.2(a)). This is done in a check-out *step. She modifies her local copy by adding the node* PhDStud *as a subtype of* Student*. These modifications take place in an* evolution *step. Since other developers may have updated the specification $\mathfrak{B}_1$, she needs to synchronise her local copy with the repository in order to merge other developers' modifications. This is done in a* synchronisation *step. However, no modifications of the specification $\mathfrak{B}_1$ have been made in the repository while Alice has been modifying it. Hence, the synchronisation is completed without changing her local copy $\mathfrak{A}_1$. Finally, Alice commits her local copy, which will be labelled $\mathfrak{B}_2$ in the repository (see Figure 3.2(b)). This is done in a* commit *step.*

Afterwards, Bob checks out a local copy $\mathfrak{B}_2$ of the specification $\mathfrak{B}_2$ from the same repository. He considers Postdoc *as a different subtype of student. To avoid the pollution of subtypes in the specification, he deletes the* PhDStud *node and refactors the specification by adding a node* Enrolment *together with the arrows* eStud *and* eUniv*. Then, he synchronises his local copy with the repository. This synchronisation is also completed without changing his local copy $\mathfrak{B}_2$. Finally,*

Bob commits his local copy, which will be labelled $\mathfrak{B}_3$ *in the repository (see Figure 3.2(c)).*

Alice continues modifying her local copy $\mathfrak{A}_2$*, which is now* out-of-date *since it is a copy of the specification* $\mathfrak{B}_2$*, while the latest specification in the repository (containing Bob's modifications) is* $\mathfrak{B}_3$*. She adds a node* Project *together with the arrows* pPhds *and* pUnivs *(see Figure 3.2(d)). Moreover, she adds the atomic constraints* (`[mult(1,4)]`, δ_1), (`[inverse]`, δ_2) *and* (`[surjective]`, δ_3) *on the arrows* sUnivs *and* uStuds*. Then, she synchronises her local copy with the repository. This time the synchronisation procedure detects conflicting modifications; e.g., Alice has added an arrow to the node* PhDStud *which Bob has deleted.*

In the following, the underlying techniques of the proposed approach to model versioning are analysed. Furthermore, several examples, built upon Example 14, are used to illustrate the application of our techniques. The notation is extended by adopting some keywords from [75]:

- *Base specification* $\mathfrak{B}_B$, with B for `BASE`: the last checked out or synchronised specification prior to any modification; i.e., the pristine version of a local copy, e.g., $\mathfrak{B}_2$ is the base specification for $\mathfrak{A}_2$
- *Head specification* $\mathfrak{B}_H$, with H for `HEAD`: the latest (or most recent) specification in the repository, e.g., $\mathfrak{B}_3$

Note that the head specification is the same for all users. In contrast, the base specification is bound to the local copy and may differ from user to user.

3.3 Calculation and representation of differences

In version control, the identification of commonalities between (versions of) artefacts is necessary to calculate their differences. For example, a solution to the longest common subsequence problem [50] is typically implemented in differencing algorithms for text-based artefacts.

HARD- VS. SOFT-LINKING

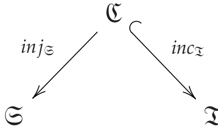
Various techniques for the identification of commonalities in MDE can be found in the literature. A rudimentary technique is based on persistent identifiers, such as Universally Unique Identifiers (UUID) [51]; in this technique, elements with equal identifiers are seen as equal elements (*hard-linking*) [73]. While this technique would work efficiently within specific tools, it is not general enough to function as a generic technique. This is because persistent identifiers are different for every environment. Another technique for the identification of commonalities is based on metrics such as structural similarity; in this technique, elements that have the same features are seen as equal elements (*soft-linking*) [59]. This technique has the benefit of being general, but it is slightly resource greedy.

RECORDING OF MODIFICATIONS

In this thesis, a different technique for the identification of commonal-

ities is proposed. Specification elements which are not modified during an evolution step are *recorded* in a *common specification*; i.e., a specification which represents the commonalities between two subsequent versions of a specification. The common specification is defined as follows:

Definition 22 (Common specification). *Given specifications* $\mathfrak{S} = (S, C^{\mathfrak{S}}: \Sigma)$ *and* $\mathfrak{T} = (T, C^{\mathfrak{T}}: \Sigma)$*, a common specification of* $\mathfrak{S}$ *and* $\mathfrak{T}$ *consists of a specification* $\mathfrak{C} := (C, C^{\mathfrak{C}}: \Sigma)$*, an injective specification morphism* $inj_{\mathfrak{S}} : \mathfrak{C} \to \mathfrak{S}$ *and an inclusion specification morphism* $inc_{\mathfrak{T}} : \mathfrak{C} \hookrightarrow \mathfrak{T}$.



In this work, the contribution of common specifications is twofold:

- For each pair of specifications $\mathfrak{V}_i$ and $\mathfrak{V}_{i+1}$, a common specification $\mathfrak{C}_{i,i+1}$ of $\mathfrak{V}_i$ and $\mathfrak{V}_{i+1}$ is stored in the repository. $\mathfrak{C}_{i,i+1}$ is called *the* common specification of $\mathfrak{V}_i$ and $\mathfrak{V}_{i+1}$ (see Figure 3.3(a)).
- For each pair of base specification $\mathfrak{V}_B$ and local copy $\mathfrak{U}_B$, a local common specification $\mathfrak{U}\mathfrak{C}_B$ of $\mathfrak{V}_B$ and $\mathfrak{U}_B$ is maintained by the VCS. $\mathfrak{U}\mathfrak{C}_B$ is called *the* common specification of $\mathfrak{V}_B$ and $\mathfrak{U}_B$ (see Figure 3.3(b)).

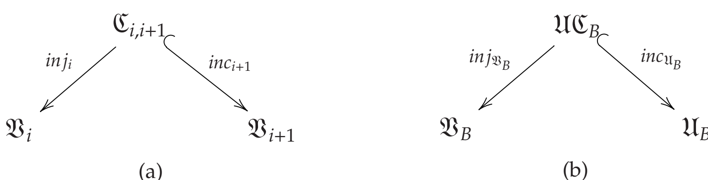


Figure 3.3: (a) The common specification $\mathfrak{C}_{i,i+1}$ of the specifications $\mathfrak{V}_i$ and $\mathfrak{V}_{i+1}$; (b) The local common specification $\mathfrak{U}\mathfrak{C}_B$ of the specifications $\mathfrak{V}_B$ and $\mathfrak{U}_B$

Note that the specification morphism $inj_{\mathfrak{S}} : \mathfrak{C} \to \mathfrak{S}$ is injective in order to allow for renaming. Moreover, the specification morphism $inc_{\mathfrak{T}} : \mathfrak{C} \hookrightarrow \mathfrak{T}$ is inclusion so that common specifications always contain the new names. An illustration of renaming is presented in Example 17.

The following example illustrates the usage of common specifications.

Example 15 (Common specification)**.** *Building upon Example 14, Figure 3.4(c) shows the common specification* $\mathfrak{C}_{2,3}$ *for the specifications* $\mathfrak{V}_2$ *and* $\mathfrak{V}_3$*.*

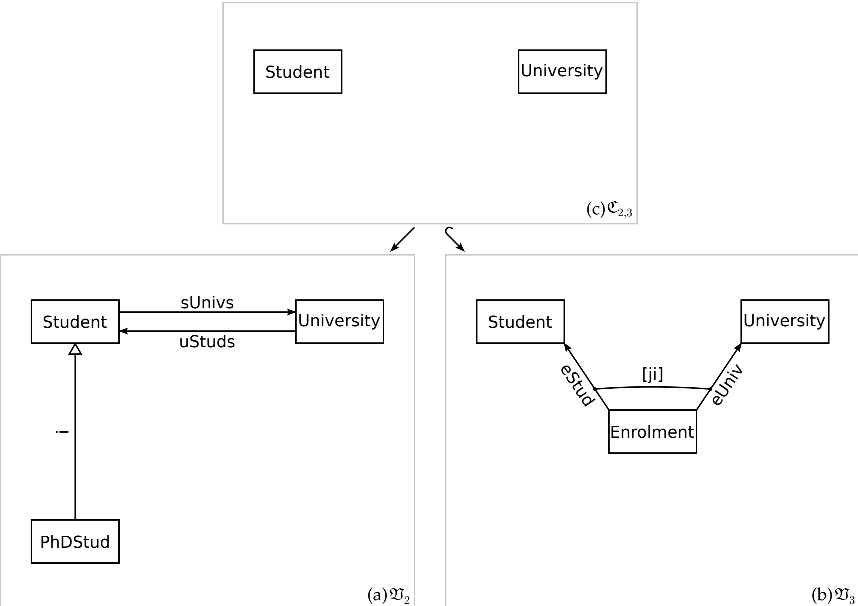


Figure 3.4: The common specification $\mathfrak{C}_{2,3}$ of the specifications $\mathfrak{V}_2$ and $\mathfrak{V}_3$

As mentioned, the identification of commonalities is necessary in order to calculate the differences between artefacts. The *calculation* and *representation* of differences focuses on identifying the modifications which have taken place in an evolution step.

Terminology

Various techniques for the calculation and representation of differences in MDE can be found in the literature [22, 59, 73, 77]. These techniques differ in that they analyse the modifications which a specification undergoes; e.g., *change* or *update* are given different and ambiguous semantics. Moreover, the terminology in these techniques is not consistent; e.g., the terms "add", "create" and "insert" are frequently used to refer to the same modification. In this work, modifications are classified as in Table 3.2.

Calculation of differences

The calculation of the difference between two subsequent versions of a specification, i.e., the information about which elements are common, added, deleted and renamed, requires the comparison of the old and the new version with their common specification. For example, all the nodes and arrows which are present in the new version but not in the common specification are identified as added. Similarly, all the nodes and arrows

Table 3.2: The classification of the modifications

Term	Definition	Alternative terms
add	an element is added to a specification	*create*, *insert*
delete	an element is deleted from a specification	*remove*
rename	an element is renamed in a specification	special case of *change* or *update*

which are present in the old version but not in the common specification are identified as deleted. In this work, the difference between two subsequent versions of a specification is implicitly given by a span of common, old and new specifications. In addition, the same information is explicitly represented by a *difference specification*; i.e., a specification which contains all common, added, deleted and renamed elements. The underlying graph and the set of atomic constraints of the difference specification are obtained by the pushout construction [12, 38], which can be regarded as a slight generalisation of union – since only injective and inclusion morphisms are considered in this thesis. The methodological motivation behind adopting difference specifications in addition to spans of common, old and new specifications is that, as will be clear later, gathering all these elements in one specification facilitates the application of transformation rules to automatically detect and resolve conflicts.

Representation of differences

Due to the diagrammatic nature of specifications, the representation of differences such as added, deleted and renamed is expressed by a diagrammatic language. The diagrammatic language for the representation of differences is given by a *tag signature* Δ, which has the same structure of a signature but no semantic counterpart. A tag signature $\Delta = (\Theta^{\Delta}, \alpha^{\Delta})$ consists of a set of *tags* $\theta \in \Theta^{\Delta}$, each having an arity $\alpha^{\Delta}(\theta)$ and a proposed visualisation. In particular, the set of tags $\Theta^{\Delta} = \Theta^{G} \cup \Theta^{\Sigma}$ consists of the union of two sets Θ^{G} and Θ^{Σ}.

The set Θ^{G} is fixed and consists of tags for annotating nodes and arrows of the underlying graph of a specification (see Table 3.3).

The set Θ^{Σ} is generated from the signature Σ (see Table 3.1) and consists of tags for annotating atomic constraints specified by means of predicates of the signature Σ (see Table 3.4). The generation of tags is defined as follows:

Table 3.3: The subset of the signature Δ for the annotation of the underlying graph

$\theta \in \Theta^G$	$\alpha^\Delta(\theta)$	Proposed visual.	Alternative visual.
<add>^{N}	1	X	X <A>
<delete>^{N}	1	X	X <D>
<rename(X,Y)>^{N}	1	Y <R:X ↦ Y>	
<conflict>^{N}	1	X	X <C>
<add>^{A}	$1 \xrightarrow{a} 2$	X —f→ Y	X —<A>f→ Y
<delete>^{A}	$1 \xrightarrow{a} 2$	X —f→ Y	X —<D>f→ Y
<rename(f,g)>^{A}	$1 \xrightarrow{a} 2$	X —g→ Y <R:f↦g>	
<conflict>^{A}	$1 \xrightarrow{a} 2$	X —f→ Y	X —<C>f→ Y

Definition 23 (Generation of Θ^Σ)**.** *Given a signature $\Sigma = (\Pi^\Sigma, \alpha^\Sigma)$, the set of tags Θ^Σ is constructed as follows:*

$$\begin{aligned}\Theta^\Sigma := \; & \{\texttt{<add>}^\pi \text{ with } \alpha^\Delta(\texttt{<add>}^\pi) = \alpha^\Sigma(\pi) \mid \pi \in \Pi^\Sigma\} \\ \bigcup \; & \{\texttt{<delete>}^\pi \text{ with } \alpha^\Delta(\texttt{<delete>}^\pi) = \alpha^\Sigma(\pi) \mid \pi \in \Pi^\Sigma\} \\ \bigcup \; & \{\texttt{<conflict>}^\pi \text{ with } \alpha^\Delta(\texttt{<conflict>}^\pi) = \alpha^\Sigma(\pi) \mid \pi \in \Pi^\Sigma\}\end{aligned}$$

Note that the tag <conflict> is not used in the difference specifications of two subsequent specifications; it is used to annotate conflicting modifications in the synchronisation procedure (see Section 3.4).

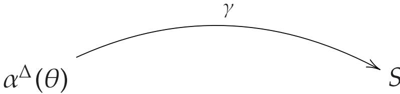
Remark 6 (Multiple visualisations)**.** *Two visualisations for the tags in Δ are proposed. The default visualisation is based on colour-coding while the alternative visualisation is based on labels. In this work, colour-coding is preferred over labels since it is the authors' experience that colouring makes it easier to understand modifications. However, labels can be adopted in case of black and white printing.*

Table 3.4: The subset of the signature Δ for the annotation of atomic constraints

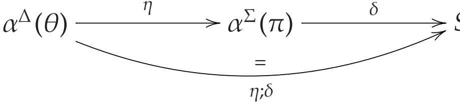
$\theta \in \Theta^{\Sigma}$	$\alpha^{\Delta}(\theta)$	Proposed visual.	Alternative visual.
<add>$^{[\texttt{mult}(m,n)]}$	$1 \xrightarrow{a} 2$	X —f→ Y [m..n]	X —f→ Y <A>[m..n]
<delete>$^{[\texttt{mult}(m,n)]}$	$1 \xrightarrow{a} 2$	X —f→ Y [m..n]	X —f→ Y <D>[m..n]
<conflict>$^{[\texttt{mult}(m,n)]}$	$1 \xrightarrow{a} 2$	X —f→ Y [m..n]	X —f→ Y <C>[m..n]
<add>$^{[\texttt{injective}]}$	$1 \xrightarrow{a} 2$	X —f→ Y [inj]	X —f→ Y <A>[inj]
<delete>$^{[\texttt{injective}]}$	$1 \xrightarrow{a} 2$	X —f→ Y [inj]	X —f→ Y <D>[inj]
<conflict>$^{[\texttt{injective}]}$	$1 \xrightarrow{a} 2$	X —f→ Y [inj]	X —f→ Y <C>[inj]
<add>$^{[\texttt{jointly-injective}]}$	$1 \xrightarrow{a} 2$, $1 \xrightarrow{b} 3$	X —f→ Y, X —g→ Z [ji]	X —f→ Y, X —g→ Z <A> [ji]
<delete>$^{[\texttt{jointly-injective}]}$	$1 \xrightarrow{a} 2$, $1 \xrightarrow{b} 3$	X —f→ Y, X —g→ Z [ji]	X —f→ Y, X —g→ Z <D> [ji]
<conflict>$^{[\texttt{jointly-injective}]}$	$1 \xrightarrow{a} 2$, $1 \xrightarrow{b} 3$	X —f→ Y, X —g→ Z [ji]	X —f→ Y, X —g→ Z <C> [ji]
<add>$^{[\texttt{surjective}]}$	$1 \xrightarrow{a} 2$	X —f→ Y [surj]	X —f→ Y <A>[surj]
<delete>$^{[\texttt{surjective}]}$	$1 \xrightarrow{a} 2$	X —f→ Y [surj]	X —f→ Y <D>[surj]
<conflict>$^{[\texttt{surjective}]}$	$1 \xrightarrow{a} 2$	X —f→ Y [surj]	X —f→ Y <C>[surj]
<add>$^{[\texttt{inverse}]}$	$1 \xrightarrow{a} 2$, $2 \xrightarrow{b} 1$	X —f→ Y, Y —g→ X [inv]	X —f→ Y, Y —g→ X <A> [inv]
<delete>$^{[\texttt{inverse}]}$	$1 \xrightarrow{a} 2$, $2 \xrightarrow{b} 1$	X —f→ Y, Y —g→ X [inv]	X —f→ Y, Y —g→ X <D> [inv]
<conflict>$^{[\texttt{inverse}]}$	$1 \xrightarrow{a} 2$, $2 \xrightarrow{b} 1$	X —f→ Y, Y —g→ X [inv]	X —f→ Y, Y —g→ X <C> [inv]

An *annotated specification* $\mathfrak{S} = (S, C^{\mathfrak{S}}\colon\Sigma, A^{\mathfrak{S}}\colon\Delta)$ consists of a specification $\mathfrak{S}$ together with a set of annotations $A^{\mathfrak{S}}$ which are specified by means of a tag signature Δ. A graph annotation (θ, γ) consists of a tag $\theta \in \Theta^{\Delta}$ and a graph homomorphism $\gamma : \alpha^{\Delta}(\theta) \to S$, while an atomic constraint annotation consists of a tag $\theta \in \Theta^{\Delta}$ and a graph homomorphism $\eta : \alpha^{\Delta}(\theta) \to \alpha^{\Sigma}(\pi)$.

Definition 24 (Graph annotation)**.** *Given a tag signature $\Delta = (\Theta^{\Delta}, \alpha^{\Delta})$, an annotation (θ, γ) on a graph S consists of a tag symbol $\theta \in \Theta^{\Delta}$ and a graph homomorphism $\gamma : \alpha^{\Delta}(\theta) \to S$.*



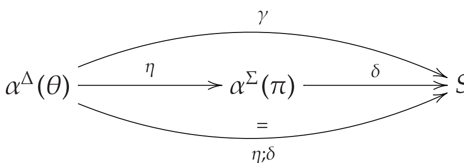
Definition 25 (Atomic constraint annotation)**.** *Given a signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$, a tag signature $\Delta = (\Theta^{\Delta}, \alpha^{\Delta})$ and a graph S, an annotation $(\theta, \eta; \delta)$ on an atomic constraint (π, δ) with $\delta : \alpha^{\Sigma}(\pi) \to S$ consists of a tag symbol $\theta \in \Theta^{\Delta}$ and a graph homomorphism $\eta : \alpha^{\Delta}(\theta) \to \alpha^{\Sigma}(\pi)$ such that the following diagram commutes:*



Definition 26 (Annotated specification)**.** *Given a signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$ and a tag signature $\Delta = (\Theta^{\Delta}, \alpha^{\Delta})$, an annotated specification $\mathfrak{S} = (S, C^{\mathfrak{S}}\colon\Sigma, A^{\mathfrak{S}}\colon\Delta)$ is a specification $\mathfrak{S}$ together with a set $A^{\mathfrak{S}}$ of:*

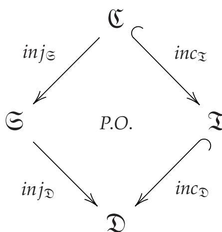
- *graph annotations (θ, γ) on S and*
- *atomic constraint annotations $(\theta, \eta; \delta)$ on $C^{\mathfrak{S}}$*

with $\theta \in \Theta^{\Delta}$, $\gamma : \alpha^{\Delta}(\theta) \to S$, $\eta : \alpha^{\Delta}(\theta) \to \alpha^{\Sigma}(\pi)$ and $(\pi, \delta) \in C^{\mathfrak{S}}$.



Considering the calculation and representation approaches described above, the difference specification is defined as follows:

Definition 27 (Difference specification)**.** *Given a common specification $\mathfrak{C}$ of specifications $\mathfrak{S}$ and $\mathfrak{T}$, the difference specification of $\mathfrak{S}$ and $\mathfrak{T}$ consists of an annotated specification $\mathfrak{D} := (D, C^{\mathfrak{D}}{:}\Sigma, A^{\mathfrak{D}}{:}\Delta)$, an injective specification morphism $inj_{\mathfrak{D}} : \mathfrak{S} \to \mathfrak{D}$ and an inclusion specification morphism $inc_{\mathfrak{D}} : \mathfrak{T} \hookrightarrow \mathfrak{D}$, where $(D, C^{\mathfrak{D}}{:}\Sigma)$ is constructed as the pushout $(\mathfrak{D},\ inj_{\mathfrak{D}} : \mathfrak{S} \to \mathfrak{D},\ inc_{\mathfrak{D}} : \mathfrak{T} \hookrightarrow \mathfrak{D})$ of the span $\mathfrak{S} \xleftarrow{inj_{\mathfrak{S}}} \mathfrak{C} \xhookrightarrow{inc_{\mathfrak{T}}} \mathfrak{T}$ in the category* **Spec**(Σ)*, according to Proposition 7.*



In addition, the set of annotations $A^{\mathfrak{D}}$ is constructed as follows:

$$
\begin{array}{rl}
A^{\mathfrak{D}} := & \{(\texttt{<add>}^{N}, \gamma) \text{ with } \gamma(\alpha^{\Delta}(\texttt{<add>}^{N})) = X \mid X \in T_N \setminus C_N\} \\
\bigcup & \{(\texttt{<add>}^{A}, \gamma) \text{ with } \gamma(\alpha^{\Delta}(\texttt{<add>}^{A})) = f \mid f \in T_A \setminus C_A\} \\
\bigcup & \{(\texttt{<delete>}^{N}, \gamma) \text{ with } \gamma(\alpha^{\Delta}(\texttt{<delete>}^{N})) = X \mid X \in S_N \setminus C_N\} \\
\bigcup & \{(\texttt{<delete>}^{A}, \gamma) \text{ with } \gamma(\alpha^{\Delta}(\texttt{<delete>}^{A})) = f \mid f \in S_A \setminus C_A\} \\
\bigcup & \{(\texttt{<rename}(inj_{\mathfrak{S}}(Y), Y)\texttt{>}^{N}, \gamma) \text{ with} \\
 & \gamma(\alpha^{\Delta}(\texttt{<rename}(inj_{\mathfrak{S}}(Y), Y)\texttt{>}^{N})) = Y \mid Y \in S_N \text{ and } Y \neq inj_{\mathfrak{S}}(Y)\} \\
\bigcup & \{(\texttt{<rename}(inj_{\mathfrak{S}}(g), g)\texttt{>}^{A}, \gamma) \text{ with} \\
 & \gamma(\alpha^{\Delta}(\texttt{<rename}(inj_{\mathfrak{S}}(g), g)\texttt{>}^{A})) = g \mid g \in S_A \text{ and } g \neq inj_{\mathfrak{S}}(g)\} \\
\bigcup & \{(\texttt{<add>}^{\pi}, \eta; \delta) \text{ with} \\
 & \eta(\alpha^{\Delta}(\texttt{<add>}^{\pi})) = \alpha^{\Sigma}(\pi) \mid (\pi, \delta) \in C^{\mathfrak{T}} \setminus C^{\mathfrak{C}}\} \\
\bigcup & \{(\texttt{<delete>}^{\pi}, \eta; \delta) \text{ with} \\
 & \eta(\alpha^{\Delta}(\texttt{<delete>}^{\pi})) = \alpha^{\Sigma}(\pi) \mid (\pi, \delta) \in C^{\mathfrak{S}} \setminus C^{\mathfrak{C}}\}
\end{array}
$$

The following example illustrates the usage of difference specifications.

Example 16 (Difference specification)**.** *Building upon Example 14, Figure 3.5(d) shows the difference specification $\mathfrak{D}$ for the specifications $\mathfrak{V}_2$ and $\mathfrak{V}_3$.*

The nodes Enrolment*, the arrows* eStud *and* eUniv*, and the atomic constraint* $(\texttt{[jointly-injective]}, \delta_4)$ *have been added to the specification $\mathfrak{V}_3$. These elements are annotated with the tag* <add> *in the difference specification $\mathfrak{D}$. This annotation is visualised by green colouring (or as a label* <A> *if adopting the alternative visualisation of Δ).*

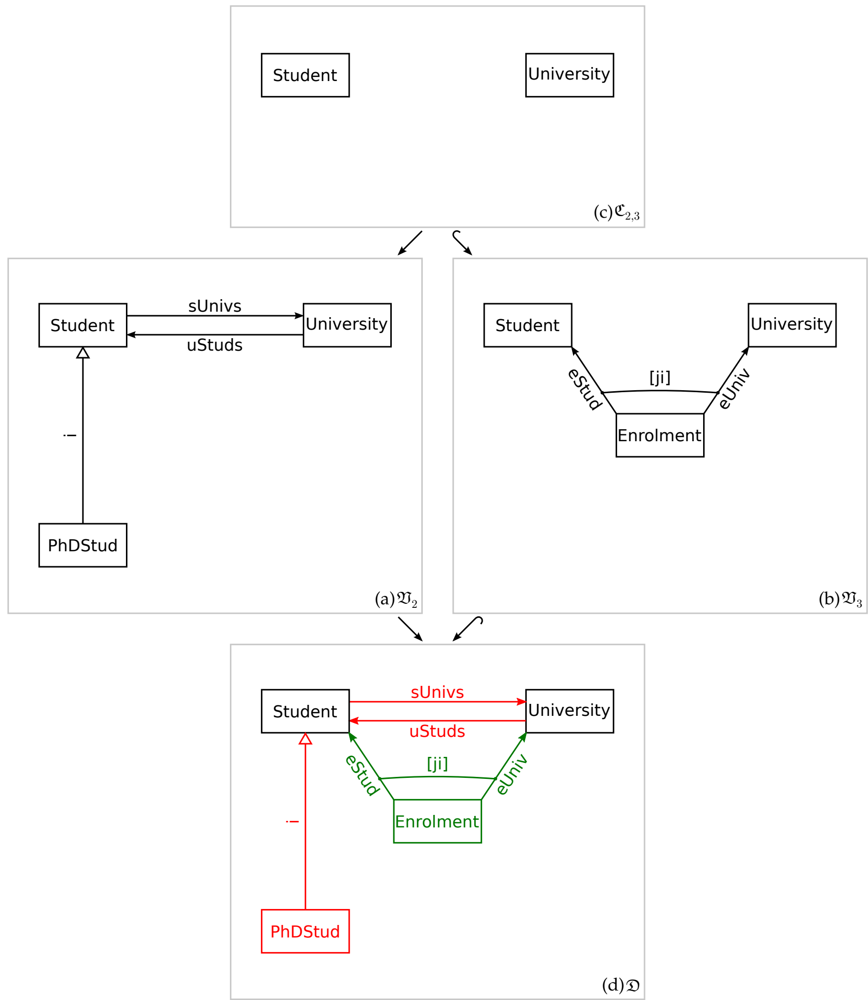


Figure 3.5: The difference specification $\mathfrak{D}$ for the specifications $\mathfrak{V}_2$ and $\mathfrak{V}_3$

With regard to the <rename(old, new)> tag, once a node (arrow) $X \in \mathfrak{S}$ is renamed to $Y \in \mathfrak{T}$, the common specification $\mathfrak{C}$ and the difference specification $\mathfrak{D}$ will contain Y with $inj_{\mathfrak{S}}(Y) = X$, $inc_{\mathfrak{T}}(Y) = Y$, $inj_{\mathfrak{D}}(X) = Y$ and $inc_{\mathfrak{D}}(Y) = Y$. Moreover, the node (arrow) Y will be annotated with the tag <rename(X, Y)>.

Recall that the specification morphisms $inj_{\mathfrak{S}} : \mathfrak{C} \rightarrow \mathfrak{S}$ and $inj_{\mathfrak{D}} : \mathfrak{S} \rightarrow \mathfrak{D}$ are injective in order to allow for this renaming. Moreover, the specification morphisms $inc_{\mathfrak{T}} : \mathfrak{C} \hookrightarrow \mathfrak{T}$ and $inc_{\mathfrak{D}} : \mathfrak{T} \hookrightarrow \mathfrak{D}$ are inclusions so that common and difference specifications always contain the new names. The following example illustrates the usage of difference specifications containing a rename.

Example 17 (Difference specification and rename)**.** *Building upon Example 14 and 16, Figure 3.6(d) shows the difference specification $\mathfrak{D}$ for the specifications $\mathfrak{V}_2$ and $\mathfrak{V}_3$.*

In addition to the modifications presented in Example 16, the node Student has been renamed to Person in $\mathfrak{V}_3$. The node Person is annotated with the tag <rename(Student, Person)> in the difference specification $\mathfrak{D}$. This annotation is visualised as <R:Student $\mapsto$ Person>. The injective specification morphism $inj_{\mathfrak{V}_2} : \mathfrak{C}_{2,3} \rightarrow \mathfrak{V}_2$ contains an explicit mapping Person $\mapsto$ Student; analogously, the injective specification morphism $inj_{\mathfrak{D}} : \mathfrak{V}_2 \rightarrow \mathfrak{D}$ contains an explicit mapping Student $\mapsto$ Person.

3.4 Synchronisation

To enable concurrent development, a mechanism for specification synchronisation is necessary. In this section, a synchronisation procedure is presented. This synchronisation procedure exploits the identification of commonalities and the calculation/representation of differences presented in the previous section.

Whenever a local copy $\mathfrak{U}_B$ is to be synchronised with the head specification $\mathfrak{V}_H$ from the repository, two cases are considered:

- If nobody has updated the head specification $\mathfrak{V}_H$; i.e., if the head specification $\mathfrak{V}_H$ and the base specification $\mathfrak{V}_B$ are identical, then the local copy is not affected by the synchronisation procedure.
- If someone has updated the head specification $\mathfrak{V}_H$; i.e., if the head specification $\mathfrak{V}_H$ and the base specification $\mathfrak{V}_B$ are different, then the modifications made by others will be merged into the local copy and possible conflicts will be detected.

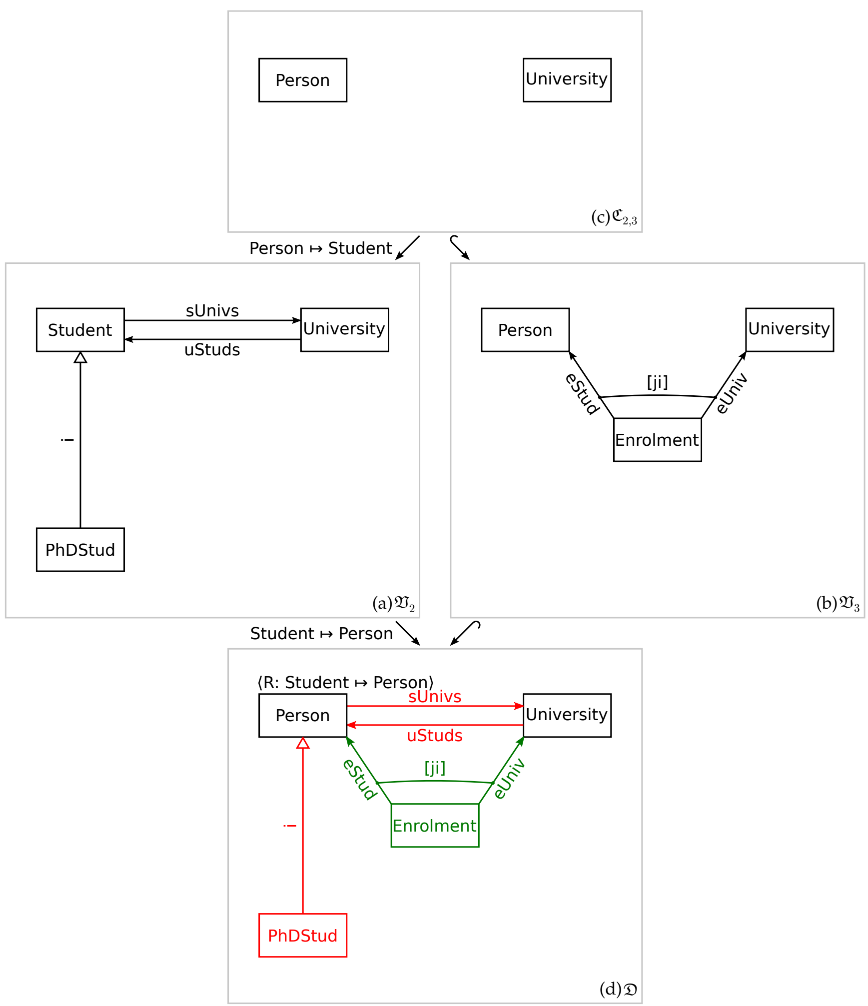


Figure 3.6: The difference specification $\mathfrak{D}$ for the specifications $\mathfrak{V}_2$ and $\mathfrak{V}_3$, containing a renaming

The synchronisation procedure takes as input the following specifications:

- The local copy $\mathfrak{U}_B$ and the local common specification $\mathfrak{UC}_B$, which are stored locally.
- The head specification $\mathfrak{V}_H$, the base specification $\mathfrak{V}_B$ and their intermediate common specifications $\mathfrak{C}_{B,B+1} \ldots \mathfrak{C}_{H-1,H}$, which are stored remotely in the repository.

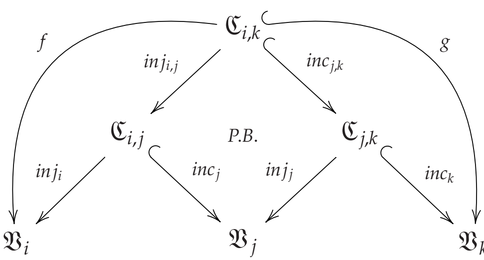
Furthermore, the synchronisation procedure is divided into the following steps:

1. Construct the *common of commons* for the base specification and the head specification.
2. Construct the difference specification for the base specification and the local copy, and the difference specification for the base specification and the head specification.
3. Construct the *merge of differences*.
4. Detect conflicts.
5. Resolve conflicts.
6. Construct the *synchronised local copy* and the *synchronised local common specification*.

3.4.1 Construct the common of commons

The common specifications stored in the repository represent the commonalities between subsequent versions of a specification. However, common specifications for specifications which are not subsequent versions of each other have to be considered as well. This is because the synchronisation procedure will construct the difference specification of the base specification $\mathfrak{V}_B$ and the head specification $\mathfrak{V}_H$ which may have an arbitrary number of intermediate specifications $\mathfrak{V}_{B+1} \ldots \mathfrak{V}_{H-1}$. This common specification, called the *common of commons*, can be constructed from the common specifications $\mathfrak{C}_{B,B+1} \ldots \mathfrak{C}_{H-1,H}$ of the intermediate specifications. One possible way to construct the common of commons is defined as follows:

Definition 28 (Common of commons)**.** *Given specifications $\mathfrak{C}_{i,j}$, $\mathfrak{C}_{j,k}$, $\mathfrak{B}_i$, $\mathfrak{B}_j$ and $\mathfrak{B}_k$, the common of commons consists of a specification $\mathfrak{C}_{i,k} := (C_{i,k}, C^{\mathfrak{C}_{i,k}}\colon \Sigma)$, an injective specification morphism $f := inj_{i,j}; inj_i$ and an inclusion specification morphism $g := inc_{j,k}; inc_k$ constructed as the pullback $(\mathfrak{C}_{i,k},\ inj_{i,j} : \mathfrak{C}_{i,k} \rightarrow \mathfrak{C}_{i,j},\ inc_{j,k} : \mathfrak{C}_{i,k} \hookrightarrow \mathfrak{C}_{j,k})$ of the co-span $\mathfrak{C}_{i,j} \overset{inc_j}{\hookrightarrow} \mathfrak{B}_j \overset{inj_j}{\longleftarrow} \mathfrak{C}_{j,k}$ in the category* **Spec**(Σ)*, according to Proposition 5.*



For numbers i, k with $(k - i) > 2$, there are different possible ways to construct a corresponding common of commons by a sequence of pullback constructions. However, all these different sequences will produce the same result, as discussed in Remark 11. Thus, one can talk about *the* common of commons and use the notation $\mathfrak{C}_{i,k}$ for this. The following example illustrates the usage of common of commons.

Example 18 (Common of commons)**.** *Building upon Example 14, Figure 3.7(f) shows the common of commons $\mathfrak{C}_{1,3}$ of the common specifications $\mathfrak{C}_{1,2}$ and $\mathfrak{C}_{2,3}$, which is the common specification for the specifications $\mathfrak{B}_1$ and $\mathfrak{B}_3$.*

Remark 7 (Identities of elements)**.** *For all i, k such that $i < k$, elements which are deleted in $\mathfrak{B}_i$ and added to $\mathfrak{B}_k$ are considered distinct elements even if they have the same name. For example, a node* Student *which is deleted from $\mathfrak{B}_1$ and a node* Student *which is added in $\mathfrak{B}_{10}$ are distinct nodes and they will not be identified in the common specification $\mathfrak{C}_{1,10}$. Similarly, elements which are added by different users are considered distinct elements even if they have the same name.*

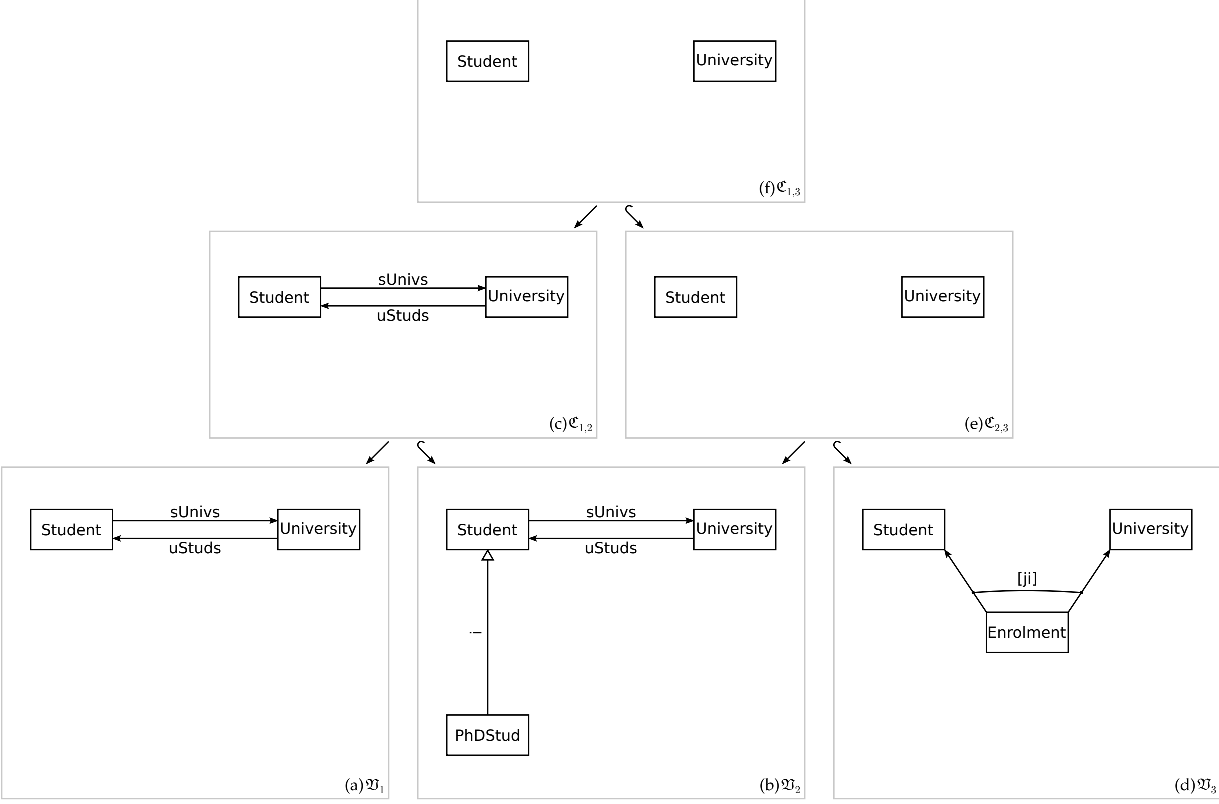
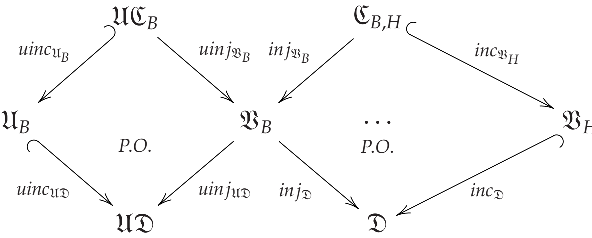


Figure 3.7: The common specification $\mathfrak{C}_{1,3}$ of the common specifications $\mathfrak{C}_{1,2}$ and $\mathfrak{C}_{2,3}$

3.4.2 Construct the difference specifications

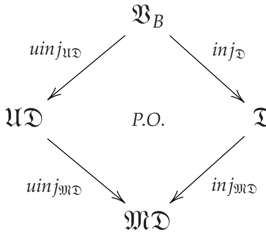
Once the common of commons $\mathfrak{C}_{B,H}$ is available, the difference specifications $\mathfrak{UD}$ and $\mathfrak{D}$ of $\mathfrak{V}_B$, $\mathfrak{U}_B$ and $\mathfrak{V}_B$, $\mathfrak{V}_H$, respectively, are constructed, according to Definition 27.



3.4.3 Construct the merge of differences

Once the difference specifications $\mathfrak{UD}$ and $\mathfrak{D}$ are available, they are merged in the merge of differences $\mathfrak{MD}$. The merge of differences is defined as follows:

Definition 29 (Merge of differences)**.** *Given specifications $\mathfrak{UD}$, $\mathfrak{D}$ and $\mathfrak{V}_B$, the merge of differences consists of a specification $\mathfrak{MD} := (MD, C^{\mathfrak{M}D}\!:\!\Sigma, A^{\mathfrak{M}D}\!:\!\Delta)$ and injective specification morphisms $uinj_{\mathfrak{MD}} : \mathfrak{UD} \to \mathfrak{MD}$ and $inj_{\mathfrak{MD}} : \mathfrak{D} \to \mathfrak{MD}$, constructed as the pushout $(\mathfrak{MD},\ uinj_{\mathfrak{MD}} : \mathfrak{UD} \to \mathfrak{MD},\ inj_{\mathfrak{MD}} : \mathfrak{D} \to \mathfrak{MD})$ of the span $\mathfrak{UD} \xleftarrow{uinj_{\mathfrak{UD}}} \mathfrak{V}_B \xrightarrow{inj_{\mathfrak{D}}} \mathfrak{D}$ in the category* **Spec**$(\Sigma \cup \Delta)$*, according to Proposition 8.*



The sets of annotations $A^{\mathfrak{U}D}$ and $A^{\mathfrak{D}}$ are merged into $A^{\mathfrak{M}D}$ by the pushout construction. While some of these annotations are identified (see Remark 13), some elements may be annotated with a pair of tags from Δ. However, only some combinations are possible. This is justified as follows:

- It is impossible to have an annotation (<add>$^{N/A}$, γ) together with any other annotation on the same node or arrow in $\mathfrak{MD}$ because elements added by different users are considered distinct even if they have the same name (see Remark 7).

- It is impossible to have two annotations (<delete>$^{N/A}$, γ) on the same node or arrow in $\mathfrak{MD}$ because they are identified by the pushout construction (see Proposition 8).

UD \ D	<add>$^{N/A}$	<delete>$^{N/A}$	<rename(old, new)>$^{N/A}$
<add>$^{N/A}$	Impossible	Impossible	Impossible
<delete>$^{N/A}$	Impossible	Identified	Possible
<rename(old, new)>$^{N/A}$	Impossible	Possible	Possible

- It is impossible to have two annotations (<add>$^{\pi}$, $\eta; \delta$) on the same atomic constraint in $\mathfrak{MD}$ because they are identified by the pushout construction (see Proposition 8).
- It is impossible to have two annotations (<delete>$^{\pi}$, $\eta; \delta$) on the same atomic constraint in $\mathfrak{MD}$ because they are identified by the pushout construction (see Proposition 8).

$C^{\mathfrak{UD}}$ \ $C^{\mathfrak{D}}$	<add>$^{\pi}$	<delete>$^{\pi}$
<add>$^{\pi}$	Identified	Impossible
<delete>$^{\pi}$	Impossible	Identified

3.4.4 Detect conflicts

The merge of differences $\mathfrak{MD}$ is then processed in order to detect conflicts. In this work, two kinds of conflicts are distinguished, namely *standard conflict* and *custom conflict*. A standard conflict occurs when concurrent modifications compete on the same elements of a specification, while a custom conflict occurs when concurrent modifications lead to semantic inconsistencies.

Standard and custom conflicts are specified by *conflict detection rules*. A conflict detection rule consists of a non-deleting transformation rule, where the left-hand side $\mathfrak{L}$ is a specification representing the conflict, and the right-hand side $\mathfrak{R}$ is a specification where the conflicting elements are annotated. The interface $\mathfrak{K}$ is equal to $\mathfrak{L}$ since non-deleting transformation rules do not delete any specification elements. The conflict detection rule is defined as follows:

Definition 30 (Conflict detection rule)**.** *A conflict detection rule consists of a transformation rule* $c = \mathfrak{L} \xleftarrow{l} \mathfrak{K} \xhookrightarrow{r} \mathfrak{R}$ *, where* $\mathfrak{L} = \mathfrak{K}$ *(see Definition 15).*

Detecting a conflict consists of applying a conflict detection rule by finding a match for the left-hand side $\mathfrak{L}$ in the merge of differences $\mathfrak{MD}$, leading to a target merge of differences $\mathfrak{MD}''$[2] where the conflicting elements are annotated. Hence, $\mathfrak{MD}$ is processed by applying all conflict detection rules, as follows:

$$\mathfrak{MD} \overset{\text{conflict detection}}{\dashrightarrow} \mathfrak{MD}''$$

Standard conflict detection

Standard conflict detection rules are divided into two sets. The first set is fixed and consists of rules for detecting conflicts on the underlying graph of a specification:

a, *b* A node or arrow is concurrently renamed and deleted.

c, *d* A node or arrow is concurrently renamed twice.

e, *f* A node is deleted while an arrow having the same node as source or target is added (dangling arrows).

Table 3.5 shows these standard conflict detection rules.

Table 3.5: The standard conflict detection rules for detecting conflicts on the underlying graph of a specification

Rule	$\mathfrak{L} = \mathfrak{K}$	$\mathfrak{R}$
a	X <R:X↦Y>	X <C> <R:X↦Y>
b	X —f→ Y <R:f↦g>	X —f→ Y <R:f↦g>
c	X <R:X↦Y> <R:X↦Z>	X <R:X↦Y> <R:X↦Z>
d	X —f→ Y <R:f↦g><R:f↦h>	X —f→ Y <R:f↦g><R:f↦h>
e	X —f→ Y	X —f→ Y
f	X ←f— Y	X ←f— Y

The second set is generated from the signature Σ and consists of rules for detecting conflicts on atomic constraints of a specification; i.e., a part of the underlying graph is deleted while an atomic constraint having the same part as target is added (dangling atomic constraints). The generation of conflict detection rules is defined as follows:

[2]The choice of the notation $\mathfrak{MD}''$ rather than $\mathfrak{MD}'$ will become clear in Section 3.4.5.

Definition 31 (Generation of conflict detection rules)**.** *Given signatures* $\Sigma = (\Pi^\Sigma, \alpha^\Sigma)$ *and* $\Delta = (\Theta^\Delta, \alpha^\Delta)$*, a set of conflict detection rules is generated as follows:*

$$
\begin{aligned}
L = K = R \; &:= \; \alpha^\Delta(\texttt{<add>}^\pi) \mid \texttt{<add>}^\pi \in \Theta^\Delta \\
C^{\mathfrak{L}} = C^{\mathfrak{K}} = C^{\mathfrak{N}} \; &:= \; \{(\pi, \delta) \mid \delta(\alpha^\Sigma(\pi)) = L\} \\
A^{\mathfrak{L}} = A^{\mathfrak{K}} \; &:= \; \{(\texttt{<delete>}^A, \gamma_1) \mid \gamma_1(\alpha^\Delta(\texttt{<delete>}^A)) = f \in L_A\} \\
&\bigcup \; \{(\texttt{<add>}^\pi, \eta_1;\delta) \mid \eta_1;\delta(\alpha^\Delta(\texttt{<add>}^\pi)) = L\} \\
A^{\mathfrak{N}} \; &:= \; A^{\mathfrak{K}} \\
&\bigcup \; \{(\texttt{<conflict>}^A, \gamma_2) \mid \gamma_2(\alpha^\Delta(\texttt{<conflict>}^A)) = f \in L_A\} \\
&\bigcup \; \{(\texttt{<conflict>}^\pi, \eta_2;\delta) \mid \eta_2;\delta(\alpha^\Delta(\texttt{<conflict>}^\pi)) = L\}
\end{aligned}
$$

Table 3.6 shows the standard conflict detection rules which are generated from the signature Σ (see Table 3.4).

Table 3.6: The standard conflict detection rules generated from Σ for detecting conflicts on atomic constraints of a specification

Rule	$\mathfrak{L} = \mathfrak{K}$	$\mathfrak{N}$
g	X —f [m..n]→ Y	X —f [m..n]→ Y
h	X —f [inj]→ Y	X —f [inj]→ Y
i	X —f→ Y; X —g→ Z; [ji]	X —f→ Y; X —g→ Z; [ji]
j	X —f [surj]→ Y	X —f [surj]→ Y
k	X —f→ Y; Y —g→ X; [inv]	X —f→ Y; Y —g→ X; [inv]

The following example illustrates the application of standard conflict detection rules.

Example 19 (Merge of differences and standard conflict detection)**.** *Building upon Example 14, Figure 3.8(h) shows the merge of differences* $\mathfrak{MD}$ *for the difference specifications* $\mathfrak{UD}$ *and* $\mathfrak{D}$*, while Figure 3.8(i) shows the merge of differences* $\mathfrak{MD}''$ *after the application of conflict detection rules.*

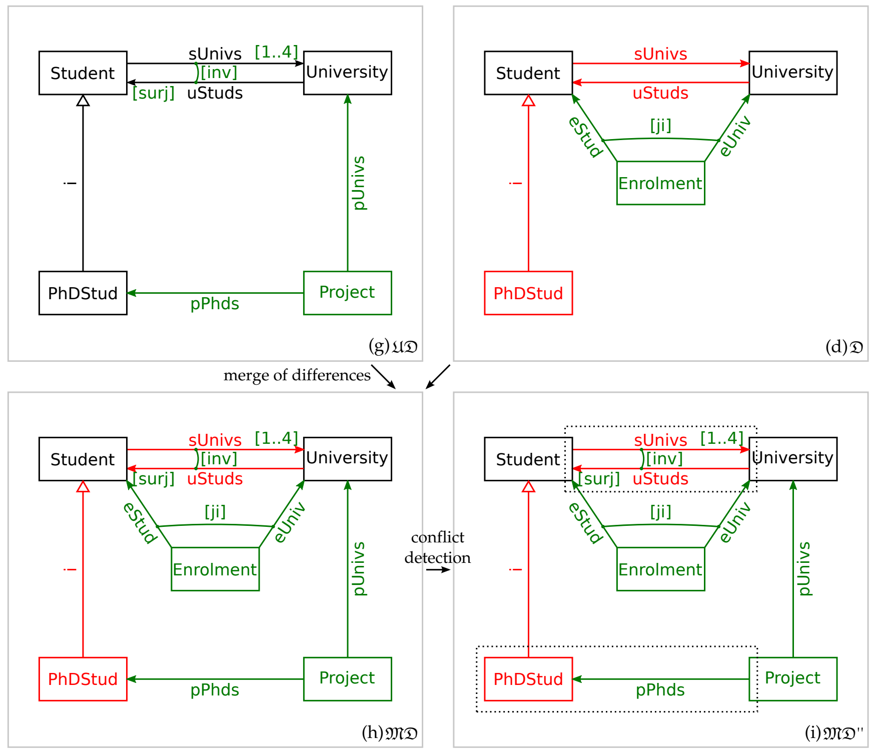
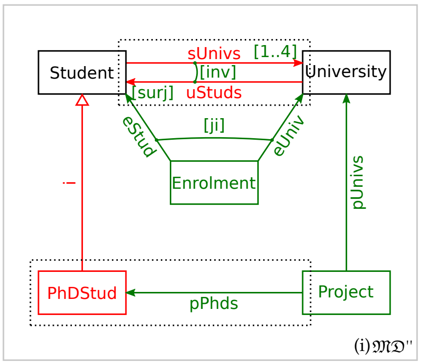


Figure 3.8: The merge of differences $\mathfrak{MD}$ and the merge of differences $\mathfrak{MD}''$ after the application of conflict detection rules

In $\mathfrak{MD}$ the node PhDStud *and the arrow* pPhds *are annotated with* `<delete>`^{N} *and* `<add>`^{A}*, respectively. In $\mathfrak{MD}''$ these nodes and arrows are additionally annotated with* `<conflict>` *according to rule f (see Table 3.6). Moreover, in $\mathfrak{MD}$ the arrows* sUnivs *and* uStuds *and the atomic constraints* (`[mult(1,4)]`, δ_1), (`[inverse]`, δ_2) *and* (`[surjective]`, δ_3) *and are annotated with* `<delete>`^{A}, `<delete>`^{A}, `<delete>`$^{[mult(m,n)]}$, `<delete>`$^{[inverse]}$ *and* `<delete>`$^{[surjective]}$*, respectively. In $\mathfrak{MD}''$ these arrows and atomic constraints are additionally annotated with* `<conflict>` *according to rules g, k and j (see Table 3.6).*

Figure 3.9(j) shows how the synchronised local copy $\mathfrak{A}_3$ would appear if constructed from the conflicting merge of differences $\mathfrak{MD}''$.

The specification $\mathfrak{A}_3$ is invalid since it contains dangling arrows and dangling atomic constraints. Note that this specification will not be constructed by the synchronisation procedure; i.e., the presence of annotations (`<conflict>`, γ) *and* (`<conflict>`, $\eta;\delta$) *in $A^{\mathfrak{MD}''}$ prevents the synchronisation procedure from creating $\mathfrak{A}_3$.*



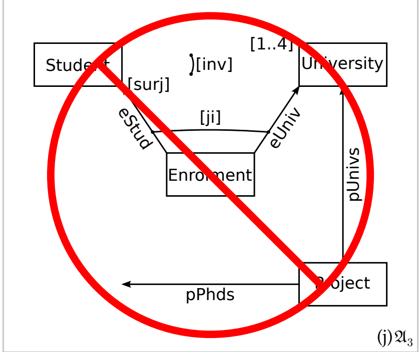


Figure 3.9: An invalid local copy $\mathfrak{A}_3$

Custom conflict detection

So far, only concurrent modifications which compete on the same elements of a specification are detected as conflicts; i.e., conflicts related to concurrent renaming as well as dangling arrows and dangling atomic constraints. However, concurrent modifications which lead to semantic inconsistencies can also be detected as conflicts. These conflicts are domain-specific and are specified as custom conflict detection rules on demand. The following example illustrates this alternative scenario of concurrent development in MDE.

Example 20 (Custom conflict detection scenario)**.** *Let us consider a variant of the scenario in Example 14. Figure 3.11 shows the different versions of the specification being developed.*

Bob and Alice check out local copies $\mathfrak{B}_2$ and $\mathfrak{A}_2$, respectively, of the specification $\mathfrak{V}_2$ from the repository. Bob adds the atomic constraint $(\texttt{[mult}(0,3)\texttt{]}, \delta_1)$ on the arrow and sUnivs*, while Alice adds the atomic constraint $(\texttt{[mult}(1,4)\texttt{]}, \delta_1)$ on the same arrow* sUnivs*. The synchronisation procedure detects conflicting modifications. This is because Alice has added a multiplicity constraint which is semantically inconsistent with the one added by Bob; i.e., according to the semantic interpretation $[\![..]\!]^{\Sigma}$ of the signature Σ (see Table 3.1), the set of valid instances of the multiplicity constraints are different. Figure 3.10(a) shows an instance which is valid for $(\texttt{[mult}(1,4)\texttt{]}, \delta_1)$ but invalid for $(\texttt{[mult}(0,3)\texttt{]}, \delta_1)$. Similarly, Figure 3.10(b) shows an instance which is valid for $(\texttt{[mult}(0,3)\texttt{]}, \delta_1)$ but invalid for $(\texttt{[mult}(1,4)\texttt{]}, \delta_1)$.*

In order to detect conflicts which are caused by concurrent modifications (or additions) of multiplicity constraints, it is possible to define a custom conflict detection rule such as the one shown in Table 3.7.

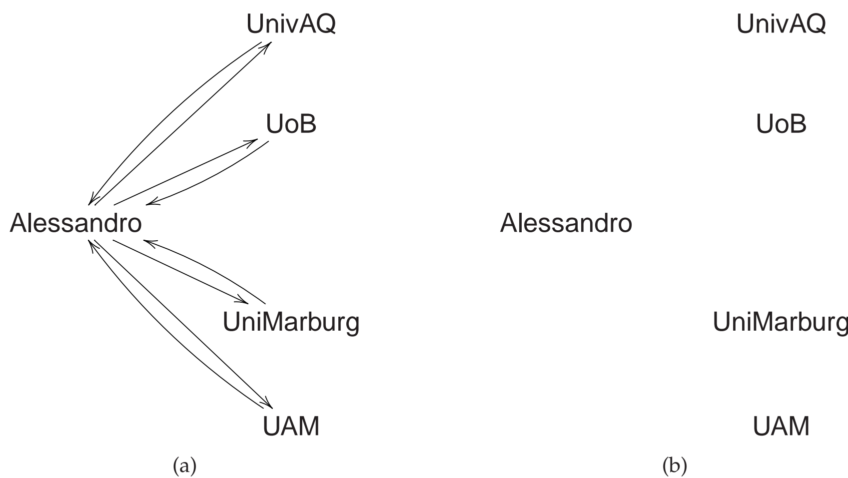


Figure 3.10: (a) An instance valid for (`[mult(1,4)]`, δ_1) but invalid for (`[mult(0,3)]`, δ_1): $|sUnivs| > 3$; (b) An instance valid for (`[mult(0,3)]`, δ_1) but invalid for (`[mult(1,4)]`, δ_1): $|sUnivs| < 1$

Table 3.7: The custom conflict detection rule for conflicts on multiplicity constraints

Rule	$\mathfrak{L} = \mathfrak{K}$	$\mathfrak{R}$
l	X —[$m_1..n_1$] [$m_2..n_2$], f→ Y	X —[$m_1..n_1$] [$m_2..n_2$], f→ Y

The rule l detects a conflict if two different multiplicity constraints are added to the same arrow.

The following example illustrates the application of custom conflict detection rules.

Example 21 (Merge of differences and custom conflict detection)**.** *Building upon Example 20, Figure 3.11(h) shows the merge of differences $\mathfrak{MD}$, while Figure 3.11(i) shows the merge of differences $\mathfrak{MD}''$ after the application of conflict detection rules.*

In $\mathfrak{MD}$ the atomic constraints (`[mult(0,3)]`, δ_1) and (`[mult(1,4)]`, δ_1) are annotated with <**add**>$^{[\texttt{mult}(m,n)]}$. *In $\mathfrak{MD}''$ these atomic constraints are additionally annotated with* <**conflict**>$^{[\texttt{mult}(m,n)]}$ *according to rule l (see Table 3.7).*

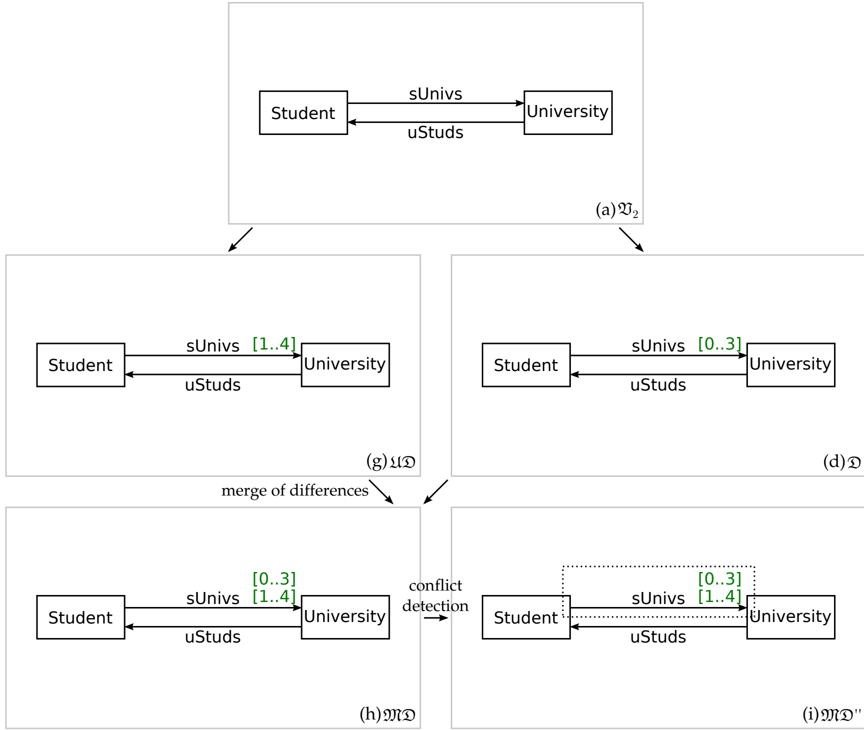


Figure 3.11: The merge of differences $\mathfrak{MD}$ and the merge of differences $\mathfrak{MD}''$ after the application of conflict detection rules

3.4.5 Resolve conflicts

Depending on the structure and semantics of the modifications, some conflicts may be automatically resolved. In this work, several *resolution strategies* [22] may be possible for a given conflict. These strategies are specified by *conflict resolution patterns*. A conflict resolution pattern consist of a transformation rule, where the left-hand side $\mathfrak{L}$ is a specification representing the conflict, the right-hand side $\mathfrak{R}$ is a specification where the resolution is applied, and $\mathfrak{K}$ is their interface. The conflict resolution pattern is defined as follows:

Definition 32 (Conflict resolution pattern)**.** *A conflict resolution pattern consists of a transformation rule* $p = \ \mathfrak{L} \xleftarrow{l} \mathfrak{K} \xhookrightarrow{r} \mathfrak{R}$.

Resolving a conflict consists of applying a conflict resolution pattern by finding a match for the left-hand side $\mathfrak{L}$ in the merge of differences $\mathfrak{MD}''$, leading to a target merge of differences $\mathfrak{MD}'''$. Hence, in addition to conflict detection rules, the merge of differences $\mathfrak{MD}$ is processed by applying all conflict resolution patterns, as follows:

$$\mathfrak{MD} \xrightarrow{\text{conflict detection}} \mathfrak{MD}'' \xrightarrow{\text{conflict resolution}} \mathfrak{MD}'''$$

As mentioned, some of the modifications which are detected as conflicts are dangling arrows. In order to resolve dangling arrows, it is possible to define two conflict resolution patterns. The first "liberal" pattern a_L is to keep the node which is targeted by the arrow. The second "conservative" pattern a_C is to remove the dangling arrow. Table 3.8 shows these conflict resolution patterns.

Table 3.8: The conflict resolution pattern for dangling arrows

Rule	$\mathfrak{L}$	$\mathfrak{K}$	$\mathfrak{R}$
a_L	X —f→ Y	X —f→ Y	X —f→ Y
a_C	X —f→ Y	X —f→ Y	X —f→ Y

Similarly, in order to resolve conflicts detected by custom conflict detection rules, such as those on multiplicity constraints, it is possible to define two conflict resolution patterns. The first "liberal" pattern b_L is to remove the conflicting multiplicity constraints and add a constraint which is the union of the two. The second "conservative" pattern b_C is to remove the conflicting multiplicity constraints and add a constraint which is the intersection of the two. Table 3.9 shows these conflict resolution patterns.

Table 3.9: The conflict resolution pattern for conflicts on multiplicity constraints

Rule	$\mathfrak{L}$	$\mathfrak{K}$	$\mathfrak{R}$
b_L	X —f→ Y $[m_1..n_1]$ $[m_2..n_2]$	X —f→ Y	X —f→ Y $[min(m_1,m_2)..max(n_1,n_2)]$
b_C	X —f→ Y $[m_1..n_1]$ $[m_2..n_2]$	X —f→ Y	X —f→ Y $[max(m_1,m_2)..min(n_1,n_2)]$

In b_L, according to the semantic interpretation $[\![\texttt{[mult(}m,n\texttt{)]}]\!]^{\Sigma}$ of the signature Σ (see Table 3.1), the set of valid instances of the atomic constraint $(\texttt{[mult(}min(m_1,m_2),max(n_1,n_2)\texttt{)]},\delta_1) \in C^{\mathfrak{R}}$ is equal to the union of the set of valid instances of the atomic constraints $(\texttt{[mult(}m_1,n_1\texttt{)]},\delta_1)$, $(\texttt{[mult(}m_2,n_2\texttt{)]},\delta_2) \in C^{\mathfrak{L}}$. This is justified as follows:

$$([\texttt{mult}(m_1,n_1)],\delta) \wedge ([\texttt{mult}(m_2,n_2)],\delta) \equiv$$

$$\equiv \begin{cases} ([\texttt{mult}(m_2,n_2)],\delta) & \text{if} \quad m_1 \le m_2 \le n_2 \le n_1 \\ ([\texttt{mult}(m_1,n_1)],\delta) & \text{if} \quad m_2 \le m_1 \le n_1 \le n_2 \\ ([\texttt{mult}(m_2,n_1)],\delta) & \text{if} \quad m_1 \le m_2 \le n_1 \le n_2 \\ ([\texttt{mult}(m_1,n_2)],\delta) & \text{if} \quad m_2 \le m_1 \le n_2 \le n_1 \end{cases}$$

Similarly, in b_C, the set of valid instances of the atomic constraint $([\texttt{mult}(max(m_1,m_2),min(n_1,n_2))],\delta_1) \in C^{\mathfrak{R}}$ is equal to the intersection of the set of valid instances of the atomic constraints $([\texttt{mult}(m_1,n_1)],\delta_1)$, $([\texttt{mult}(m_2,n_2)],\delta_2) \in C^{\mathfrak{L}}$. This is justified as follows:

$$([\texttt{mult}(m_1,n_1)],\delta) \vee ([\texttt{mult}(m_2,n_2)],\delta) \equiv$$

$$\equiv \begin{cases} ([\texttt{mult}(m_1,n_1)],\delta) & \text{if} \quad m_1 \le m_2 \le n_2 \le n_1 \\ ([\texttt{mult}(m_2,n_2)],\delta) & \text{if} \quad m_2 \le m_1 \le n_1 \le n_2 \\ ([\texttt{mult}(m_1,n_2)],\delta) & \text{if} \quad m_1 \le m_2 \le n_1 \le n_2 \\ ([\texttt{mult}(m_2,n_1)],\delta) & \text{if} \quad m_2 \le m_1 \le n_2 \le n_1 \end{cases}$$

Note that the conflict resolution patterns b_L and b_C can be applied only under the condition that the range of the multiplicity constraints overlap, i.e., if $n_1 \ge m_2$ or $m_1 \le n_2$. This could be formulated as a NAC.

The following example illustrates the application of conflict resolution patterns.

Example 22 (Conflict resolution). *Building upon Example 20, Figure 3.12(i) shows the merge of differences $\mathfrak{MD}''$, while Figures 3.12(j) and 3.12(k) show the merge of differences $\mathfrak{MD}'''$ after the application of the liberal and conservative conflict resolutions patterns, respectively.*

In $\mathfrak{MD}''$ the atomic constraints $([\texttt{mult}(0,3)],\delta_1)$ and $([\texttt{mult}(1,4)],\delta_1)$ are annotated with <add>$^{[\texttt{mult}(m,n)]}$ *and* <conflict>$^{[\texttt{mult}(m,n)]}$. *In $\mathfrak{MD}'''$ these atomic constraints are replaced with a new atomic constraint $([\texttt{mult}(0,4)],\delta_1)$, according to pattern b_L, or $([\texttt{mult}(1,3)],\delta_1)$, according to pattern b_C (see Table 3.6).*

Normalisation, Conflict Detection and Conflict Resolution

Recall that the merge of differences $\mathfrak{MD}$ is constructed as the pushout of the difference specifications $\mathfrak{UD}$ and $\mathfrak{D}$. In general, $\mathfrak{MD}$ is a valid specification by construction, but it may not be in *normal form*; i.e., single atomic constraints of the specification may hide constraints that are given by the conjunction of the atomic constraints in a specification. Performing conflict detection on a merge of differences $\mathfrak{MD}$ which is not in normal form may lead to a merge of differences $\mathfrak{MD}''$ containing false negatives [62]; i.e., containing actual conflicts which are not annotated with <conflict>.

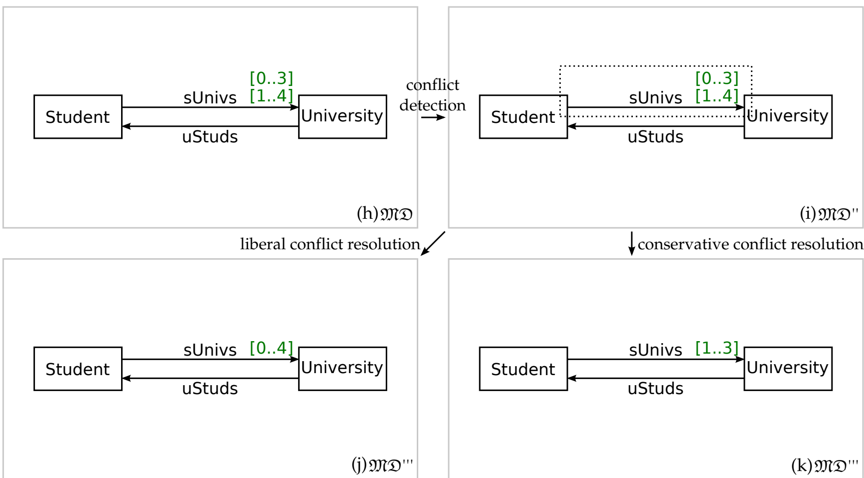


Figure 3.12: The merge of differences $\mathfrak{MD}''$ and the merge of differences $\mathfrak{MD}'''$ after the application of the conflict resolution patterns

Moreover, performing conflict resolution on the merge of differences $\mathfrak{MD}''$ which is not in normal form may lead to a merge of differences $\mathfrak{MD}'''$ which is also not in normal form. The following example illustrates this alternative scenario of concurrent development in MDE.

Example 23 (Alternative custom conflict detection scenario). *Let us consider a variant of the scenario in Example 20. Figure 3.13 shows the different versions of the specification being developed.*

In addition to the atomic constraint (`[mult(1,4)]`, δ_1) *on the arrow* sUnivs*, Alice adds the atomic constraints* (`[surjective]`, δ_3) *on the arrows* uStuds *and* (`[inverse]`, δ_2) *on the arrows* sUnivs *and* uStuds*.*

Figure 3.13(h) shows the merge of differences $\mathfrak{MD}$, Figure 3.13(j) shows the merge of differences $\mathfrak{MD}''$ after the application of conflict detection rules, while Figures 3.13(k) and 3.13(l) show the merge of differences $\mathfrak{MD}'''$ after the application of the liberal and conservative conflict resolutions patterns, respectively.

The application of the conflict resolution pattern b_L for multiplicity constraints (see Table 3.9) leads to a merge of differences $\mathfrak{MD}'''$ which is not in normal form. This is because it is possible to find a match for the left-hand side $\mathfrak{L}$ of the transformation rule $t = \mathfrak{L} \xleftarrow{l} \mathfrak{K} \xhookrightarrow{r} \mathfrak{R}$ induced by the specification entailment $e = \mathfrak{L} \vdash \mathfrak{R}$ (see Section 2.6). Indeed, the atomic constraint (`[mult(0,4)]`, δ_1) *on the arrow* sUnivs *has cardinality between zero and four (see Figure 3.13(k)), while the function sUnivs has cardinality between one and four due to the surjectivity of the inverse function uStuds. The application of the transformation rule t induced by the specification entailment e would replace the atomic constraint* (`[mult(0,4)]`, δ_1) *with* (`[mult(1,4)]`, δ_1)*, leading to the normal form of $\mathfrak{MD}'''$.*

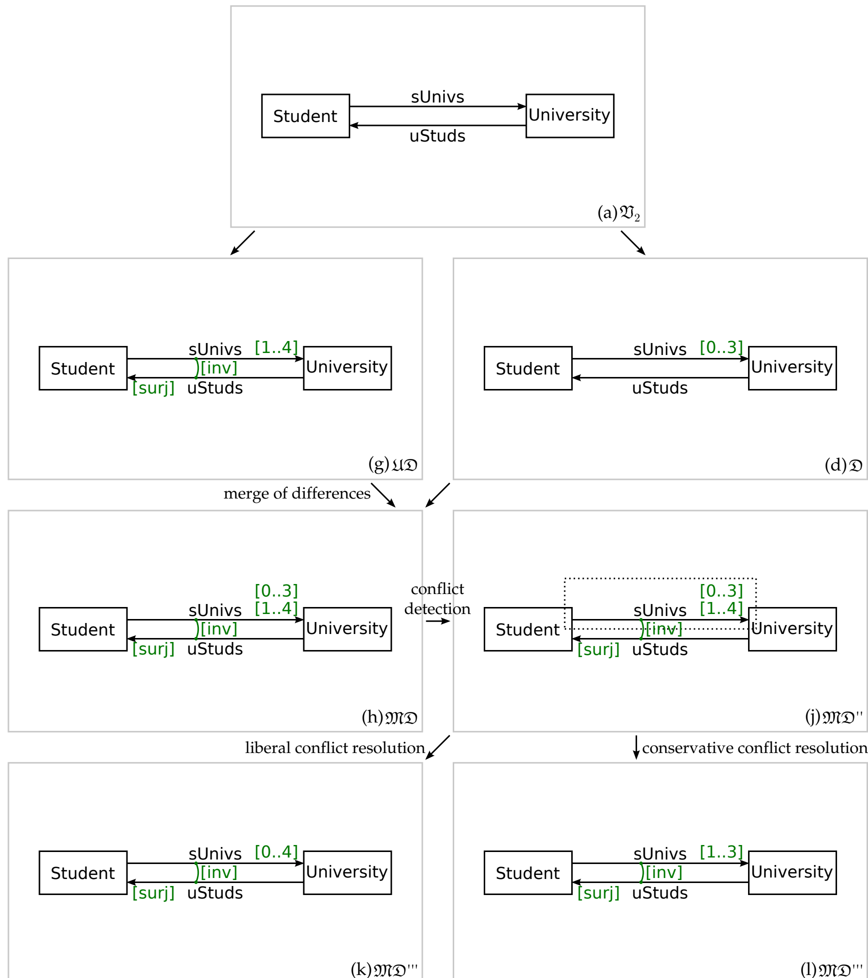


Figure 3.13: The merge of differences $\mathfrak{MD}$, the merge of differences $\mathfrak{MD}''$ after the application of conflict detection rules and the merge of differences $\mathfrak{MD}'''$ after the application of the conflict resolutions patterns

The previous example shows that single atomic constraints of a specification may hide constraints that are given by the conjunction of the atomic constraints in a specification. These hidden constraints can be made explicit by means of *normalisation* of the specification, leading to a *normal form* of the specification. In this work, normalisation consists of a sequence of applications of transformation rules induced by specification entailments. More precisely, given a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ and a set of transformation rules induced by specification entailments, a normalisation consists of a specification transformation $\mathfrak{S} \overset{*}{\Longrightarrow} \mathfrak{S}'$, leading to a normal form $\mathfrak{S}'$.

Definition 33 (Normalisation)**.** *Given a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Delta)$ and a set of transformation rules induced by specification entailments, a normalisation $\mathfrak{S} \overset{*}{\Longrightarrow} \mathfrak{S}'$ consists of a specification transformation $\mathfrak{S} \overset{*}{\Longrightarrow} \mathfrak{S}'$ where no further transformation rules are applicable to the specification $\mathfrak{S}'$.*

In this work, normalisation is assumed to be terminating and confluent; i.e., each specification can be transformed to a unique normal form by specification transformation.

Remark 8 (Termination and confluence of normalisation)**.** *The identification of the conditions under which a set of specification entailments guarantees termination and confluence of the normalisation is outside the scope of this work and will be investigated in future work (see Section 3.6).*

In order to ensure that conflict detection and conflict resolution behave as expected, they have to be performed on a merge of differences $\mathfrak{MD}$ in normal form. Hence, the process of the merge of difference has to be revised by adding normalisation before conflict detection and conflict resolution, as follows:

$$\mathfrak{MD} \xrightarrow{\text{normalisation}} \mathfrak{MD}' \xrightarrow{\text{conflict detection}} \mathfrak{MD}'' \xrightarrow{\text{conflict resolution}} \mathfrak{MD}'''$$

The following example illustrates the usage of normalisation.

Example 24 (Normalisation, conflict detection and conflict resolution)**.** *Building upon Example 23, Figure 3.14(i) shows the merge of differences $\mathfrak{MD}'$ after the normalisation, Figure 3.14(j) shows the merge of differences $\mathfrak{MD}''$ after the application of conflict detection rules, while Figures 3.14(k) and 3.14(l) show the merge of differences $\mathfrak{MD}'''$ after the application of the liberal and conservative conflict resolutions patterns, respectively.*

The normalisation replaces the atomic constraint $(\texttt{[mult(0,3)]}, \delta_1)$ in $\mathfrak{MD}$ with $(\texttt{[mult(1,3)]}, \delta_1)$ in $\mathfrak{MD}'$. As a consequence, the application of the conflict resolution pattern b_L (see Table 3.9) leads to a merge of differences $\mathfrak{MD}'''$ which is in normal form.

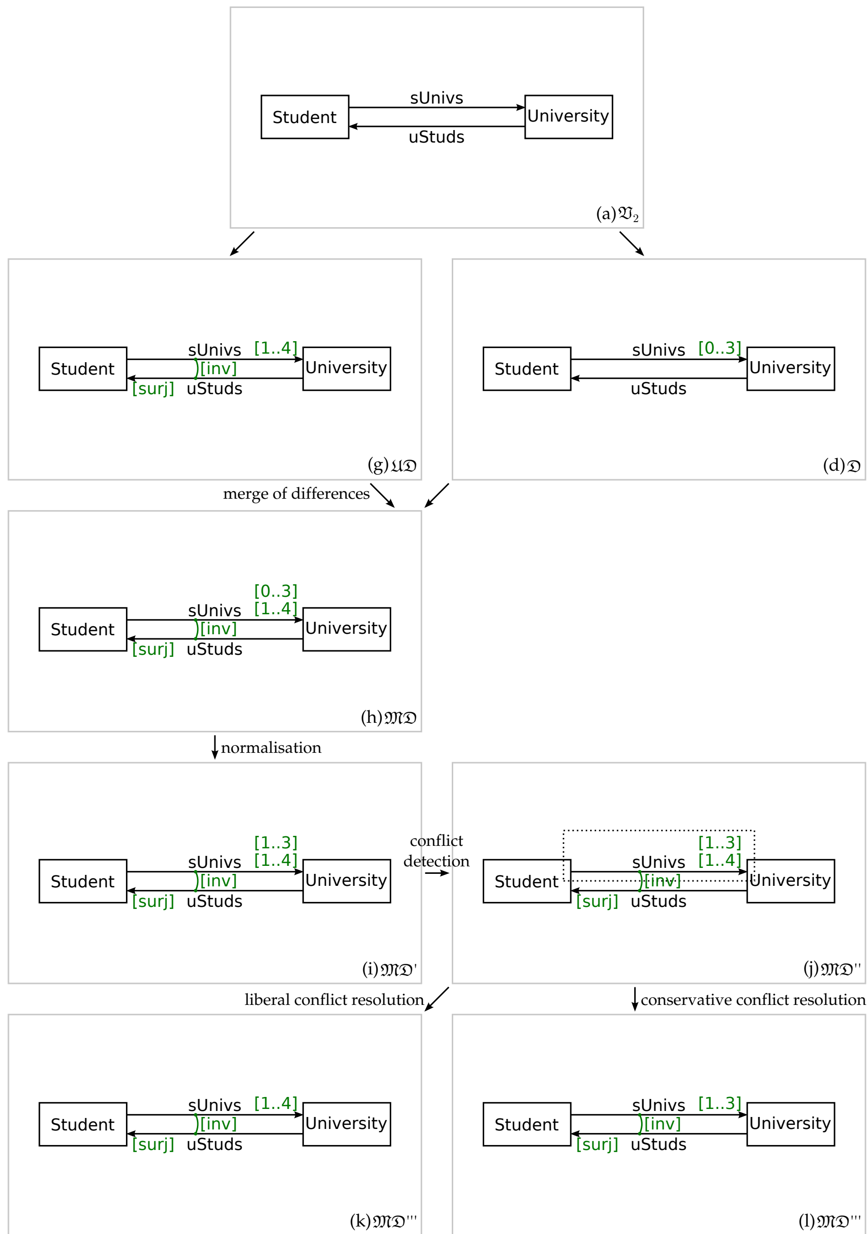


Figure 3.14: The merge of differences $\mathfrak{MD}$, the merge of differences $\mathfrak{MD}'$ after the normalisation, the merge of differences $\mathfrak{MD}''$ after the application of conflict detection rules and the merge of differences $\mathfrak{MD}'''$ after the application of the conflict resolution patterns

Remark 9 (Alternative sequences to process the merge of differences)**.** *There are alternative sequences that could be adopted to process the merge of differences $\mathfrak{MD}$. An alternative sequence could have the normalisation performed after conflict detection and resolution, as follows:*

$$\mathfrak{MD} \xrightarrow{\textit{conflict detection}} \mathfrak{MD}' \xrightarrow{\textit{conflict resolution}} \mathfrak{MD}'' \xrightarrow{\textit{normalisation}} \mathfrak{MD}'''$$

This sequence has not been adopted since conflict detection may lead to false negatives when performed on a merge of differences $\mathfrak{MD}$ which is not in normal form.

Another alternative sequence could include a loop, as follows:

$$\mathfrak{MD} \xrightarrow{\textit{conflict detection}} \mathfrak{MD}' \xrightarrow{\textit{conflict resolution}} \mathfrak{MD}'' \xrightarrow{\textit{normalisation}} \mathfrak{MD}''' \dashrightarrow \mathfrak{MD}$$

This loop may be necessary if certain conflict resolution patterns actually solve a conflict but introduce others. The conditions under which a set of conflict resolution patterns guarantees that no new conflicts are introduced is outside the scope of this work and will be investigated in future work (see Section 3.6).

3.4.6 Construct the synchronised specifications

Should the merge of differences $\mathfrak{MD}'''$ contain annotations (<conflict>, γ) or (<conflict>, $\eta; \delta$), the synchronisation procedure will stop and the developer will be asked to resolve conflicts manually. Otherwise, the synchronisation procedure will create the synchronised local copy $\mathfrak{U}_H$ and the synchronised local common specification $\mathfrak{UC}_H$. Note that while the merge of differences is an annotated specification, the synchronised local copy and and the synchronised local common specification are plain specifications; i.e., they do not have the set of annotations $A^{\mathfrak{U}_H}$ and $A^{\mathfrak{UC}_H}$, respectively.

Definition 34 (Synchronised local copy)**.** *Given a non-conflicting merge of differences $\mathfrak{MD}'''$, the synchronised local copy consists of a specification $\mathfrak{U}_H := (U_H, C^{\mathfrak{U}_H} : \Sigma)$ and an injective specification morphism $inj_{\mathfrak{MD}'''} : \mathfrak{U}_H \to \mathfrak{MD}'''$, constructed by applying the following transformation rules to $\mathfrak{MD}'''$ (see Table 3.10):*

Table 3.10: The transformation rules for extraction of synchronised local copy

Rule	$\mathfrak{L}$	$\mathfrak{K}$	$\mathfrak{R}$
ext_1	X (red)		
	X —f→ Y (red)	X Y	X Y
ext_2	X <R:X↦Y>	Y	Y
	X —f→ Y <R:f↦g>	X —g→ Y	X —g→ Y

Definition 35 (Synchronised local common specification)**.** *Given specifications $\mathfrak{V}_H$ and $\mathfrak{U}_H$, the synchronised local common specification consists of a specification $\mathfrak{UC}_H := (UC_H, C^{\mathfrak{UC}_H}: \Sigma)$, an injective specification morphism $uinj_{\mathfrak{V}_H} : \mathfrak{UC}_H \to \mathfrak{V}_H$ and an inclusion specification morphism $inc_{\mathfrak{U}_H} : \mathfrak{UC}_H \hookrightarrow \mathfrak{U}_H$, constructed as the pullback $(\mathfrak{UC}_H,\ uinj_{\mathfrak{V}_H} : \mathfrak{UC}_H \to \mathfrak{V}_H,\ inc_{\mathfrak{U}_H} : \mathfrak{UC}_H \hookrightarrow \mathfrak{U}_H)$ of the span $\mathfrak{V}_H \xrightarrow{inc_{\mathfrak{D}};inj_{\mathfrak{MD}};se;cd;cr} \mathfrak{MD}''' \xleftarrow{inj_{\mathfrak{MD}'''}} \mathfrak{U}_H$ in the category* **Spec**(Σ)*, according to Proposition 5.*

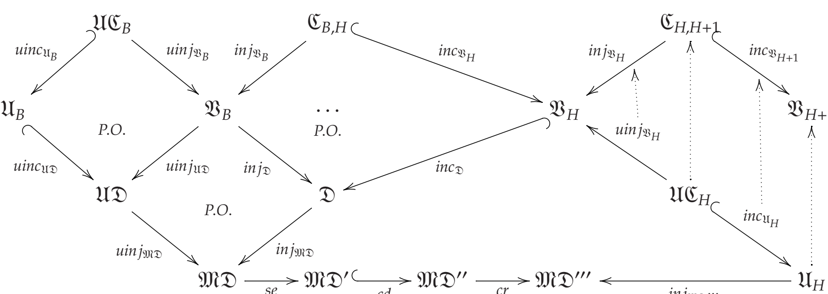


Figure 3.15: The synchronisation procedure

Finally, when all the building blocks for synchronising the local copy with the head specification are in place, the synchronisation can be fulfilled. The synchronisation is defined as follows (see Figure 3.15):

Definition 36 (Synchronisation)**.** *Given specifications $\mathfrak{U}_B$, $\mathfrak{V}_B$, $\mathfrak{V}_H$, $\mathfrak{UC}_B$ and $\mathfrak{C}_{B,B+1} \ldots \mathfrak{C}_{H-1,H}$, the synchronisation $sync : (\mathfrak{U}_B, \mathfrak{V}_B, \mathfrak{V}_H, \mathfrak{UC}_B, \mathfrak{C}_{B,B+1} \ldots \mathfrak{C}_{H-1,H}) \to (\mathfrak{U}_H, \mathfrak{UC}_H,\ inc_{\mathfrak{U}_H} : \mathfrak{UC}_H \hookrightarrow \mathfrak{U}_H,\ uinj_{\mathfrak{V}_H} : \mathfrak{UC}_H \to \mathfrak{V}_H)$ is a procedure which generates a synchronised local copy $\mathfrak{U}_H$ and a synchronised local common specification $\mathfrak{UC}_H$, according to the following procedure:*

1 *if* $B < H$
2   *given* $\mathfrak{B}_B$, $\mathfrak{U}_B$ *and* $\mathfrak{UC}_B$, *construct the difference specification* $\mathfrak{UD}$;
3   *if* $H > (B+1)$
4   *given* $\mathfrak{B}_B$, $\mathfrak{B}_H$ *and* $\mathfrak{C}_{B,B+1} \ldots \mathfrak{C}_{H-1,H}$, *construct the common of commons* $\mathfrak{C}_{B,H}$;
5   *else*
6   *the common specification* $\mathfrak{C}_{B,H}$ *is given;*
7   *given* $\mathfrak{B}_B$, $\mathfrak{B}_H$ *and* $\mathfrak{C}_{B,H}$, *construct the difference specification* $\mathfrak{D}$;
8   *given* $\mathfrak{B}_B$, $\mathfrak{UD}$ *and* $\mathfrak{D}$ *construct the merge of the differences* $\mathfrak{MD}$;
9   *given* $\mathfrak{MD}$
10   *construct* $\mathfrak{MD}'$ *by normalising* $\mathfrak{MD}$;
11   *construct* $\mathfrak{MD}''$ *by conflict detection on* $\mathfrak{MD}'$;
12   *construct* $\mathfrak{MD}'''$ *by conflict resolution on* $\mathfrak{MD}''$;
13   *if* $(\text{<conflict>},\gamma),(\text{<conflict>},\eta;\delta) \notin A^{\mathfrak{MD}'''}$
14   *given* $\mathfrak{MD}'''$ *construct the synchronised local copy* $\mathfrak{U}_H$ *and the synchronised local common specification* $\mathfrak{UC}_H$;
15   *else*
16   *display* $\mathfrak{MD}'''$;
17   *ask for manual conflict resolution;*
18 *else*
19   *the local copy is already synchronised;*

Once the synchronisation is performed, the synchronised local copy may be committed to the repository. The committed specification will be the new head specification, labelled $\mathfrak{B}_{H+1}$ in the repository. In addition, the commit will add the synchronised local common specification as the common specification of $\mathfrak{B}_H$ and $\mathfrak{B}_{H+1}$, labelled $\mathfrak{C}_{H,H+1}$ in the repository. The commit is defined as follows (see Figure 3.15):

Definition 37 (Commit)**.** *Given a synchronisation sync* : $(\mathfrak{U}_B, \mathfrak{B}_B, \mathfrak{B}_H, \mathfrak{UC}_B, \mathfrak{C}_{B,B+1} \ldots \mathfrak{C}_{H-1,H}) \rightarrow (\mathfrak{U}_H, \mathfrak{UC}_H, inc_{\mathfrak{U}_H} : \mathfrak{UC}_H \hookrightarrow \mathfrak{U}_H, uinj_{\mathfrak{B}_H} : \mathfrak{UC}_H \rightarrow \mathfrak{B}_H)$, *the commit com* : $(\mathfrak{U}_H, \mathfrak{UC}_H, inc_{\mathfrak{U}_H}, uinj_{\mathfrak{B}_H}) \dashrightarrow (\mathfrak{B}_{H+1}, \mathfrak{C}_{H,H+1}, inc_{\mathfrak{B}_{H+1}} : \mathfrak{C}_{H,H+1} \hookrightarrow \mathfrak{B}_{H+1}, inj_{\mathfrak{B}_H} : \mathfrak{C}_{H,H+1} \rightarrow \mathfrak{B}_H)$ *is an operation which adds the specifications* $\mathfrak{U}_H$ *and* $\mathfrak{UC}_H$ *to the repository as* $\mathfrak{B}_{H+1}$ *and* $\mathfrak{UC}_H$*, respectively, and the specification morphisms* $inc_{\mathfrak{U}_H}, uinj_{\mathfrak{B}_H}$ *as* $inc_{\mathfrak{B}_{H+1}}, inj_{\mathfrak{B}_H}$*, respectively.*

The following example illustrates all the steps of a synchronisation procedure.

Example 25 (Synchronisation procedure)**.** *Building upon Example 24, Figure 3.16 shows the complete execution of the synchronisation procedure.*

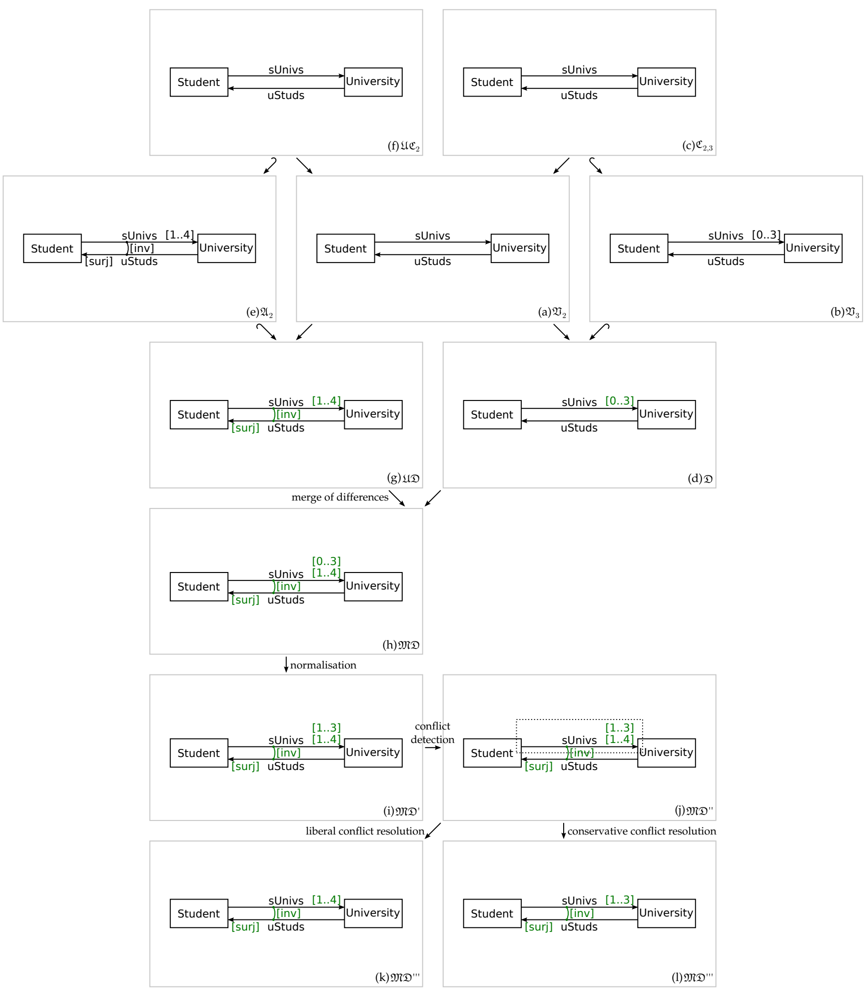


Figure 3.16: The complete execution of the synchronisation procedure

3.5 Related work

Model versioning has been greatly discussed in the literature. A first strand of research focuses on the problem of representation of differences. Three categories of representation of differences can be distinguished in the literature:

Representation of differences

- As models which conform to a difference metamodel. The difference metamodel can be generic [77], or obtained by an automated transformation [21]. These models are in general minimalistic (i.e., only the necessary information to represent the difference is presented), transformative (i.e., each difference model induces a transformation), compositional (i.e., difference models can be composed sequentially or in parallel) and typically symmetric (i.e., the inverse of a given representation of differences can be computed).

  Difference metamodel

- As a model which is the union of the two compared models, with the modified elements highlighted by colours, tags, or symbols [73]. The adoption of this technique is typically beneficial for the designer, since the rationale of the modifications is easily readable. However, these benefits apply only if the base models are not large and not too many updates apply to the same elements, since the difference model resorts to both base models to denote the differences.

  Annotations

- As a sequence of transformations describing how the initial model has been procedurally modified [1]. While this technique has the great advantage of being efficient, the representation of differences is neither readable nor intuitive. In addition, the sequence of transformations do not follow the "everything is a model vision" [16]. They are suitable for internal representations but quite ineffective for documenting modifications in MDE environments.

  Sequence of transformations

According to this classification, our representation of differences falls into the second category. The difference between models is presented in a difference model where the modified elements are annotated (and coloured to enhance readability).

A second strand of research focuses on the problem of model merging. Different formalisations can be found in the literature:

Model merging

- The work in [22] introduces a domain-specific modelling language for the definition of weaving models which represent patterns of conflicting modifications. A resolution criteria for these patterns can be specified through OCL expressions.

  Weaving models

Set theory and predicate logic

- The work in [88] presents a formal approach to the three-way merging of Ecore [86] models based on set theory and predicate logic. It is based on formally defined merge rules which can handle additions, deletions and renames of model elements and, in addition, moves of contained model elements. Moreover, it detects and resolves conflicting modifications of the same element and of different interdependent elements. Finally, the approach guarantees that the resulting merged model is a well-formed model.

Category theory and graph transformation

- The work in [87] proposes a formal approach to the merging of typed attributed graphs based on graph transformations and category theory. In this approach, two kinds of conflicts are defined based on the notion of graph modifications: operation-based and state-based conflicts. On the one hand, operation-based conflicts are detected by first extracting minimal rules from modifications and thereafter, if possible, selecting pre-defined operation rules. Conflict detection is then based on parallel dependence of graph transformations and extraction of critical pairs. On the other hand, state-based conflicts are detected by checking the merged graphs against graph constraints.

Sequence of transformations

- The work in [26] proposes a technique for obtaining automatically generated repair plans for a given inconsistent model. Repair plans are sequences of concrete modifications to be performed over a given model that fix existing inconsistencies without introducing new ones. The technique is based on Praxis, which is a model inconsistency detection approach. In Praxis, the model is represented as the sequence of actions executed by the user in order to build it.

In contrast to our approach, the above mentioned approaches do not take constraints on model elements into account. However, the approaches in [26, 87, 88] include checking the well-formedness of the result of merging. This is an important dimension of model versioning which has not been explored yet in our approach.

Heterogeneous synchronisation

A third strand of research focuses on the problem of *heterogeneous synchronisation*. In [3] the authors propose a tutorial which aims at exploring the design space of heterogeneous synchronisation. The term heterogeneous synchronisers is used by the authors to denote procedures that automate – fully or in part – the synchronisation process for (software) artefacts which are expressed in different languages. Various approaches to synchronisation of heterogeneous software artefacts are analysed and compared. In particular, the tutorial covers both the simpler synchronisation scenarios where some artefacts are never edited directly but are re-generated from other artefacts, and the more complex scenarios where several artefacts that can be modified directly need to be synchronised.

Heterogeneous modelling languages and metamodelling are important dimensions of MDE. However, in the present thesis we have not fully explored these dimensions. Our synchronisation procedure takes as input homogeneous models expressed in one modelling language. The aim of the proposed formalisation is to cover all aspects of optimistic version control and provide formal definitions of these aspects in terms of category-theoretical constructs.

Prototype tools

Finally, research has lead to a number of prototype tools that support model versioning:

DSMDiff and EMF Compare

- DSMDiff [59] and EMF Compare [37] are two model differencing tools which are based on a similar technique. Difference calculation is divided in two phases. The first focuses on model mappings, where all the elements of the two input models are compared using measures like signature matching and structural similarity. The second phase determines differences, detecting all the additions, deletions and changes. The benefit of this approach is that it is general, but this is at the price of it being slightly resource greedy.

AMOR

- In [19], the authors present AMOR, a VCS which can deal with arbitrary modelling languages based on Ecore. AMOR is built around Subversion in order to provide a centralised approach to optimistic version control, but reuses an extended version of EMF Compare for difference calculation. AMOR provides conflict detection features which may be enhanced with user-defined operations. Moreover, it provides collaborative conflict resolution features, which allow the implementation of conflict resolution policies. If the resolution is performed manually, it is analysed in order to derive resolution recommendations for similar situations which occur in future scenarios.

In contrast to our approach, the above mentioned tools do not provide a formal treatment of conflict detection and resolution. An implementation of our formalisation of model versioning is just in its initial stage of development (see Section 3.6). Once a prototype tool will be available, case studies will be performed to compare the existing tools with our tool.

3.6 Conclusion and future work

In this chapter, we described a formal approach to model versioning based on DPF. Firstly, we defined the identification of commonalities and calculation of differences as pullback and pushout constructions, respectively. Secondly, we defined the representation of differences; i.e., the information added, deleted and renamed, as a set of annotations which are specified by means of a tag signature. Thirdly, we introduced a synchronisation procedure which includes normalisation, conflict detection and conflict resolution. Specification entailments are adopted to describe properties of the semantic interpretation of predicates of a signature. The normalisation of a specification is then formalised as the embedding of these specification entailments to obtain the normal form of a specification. Moreover, transformation rules are used to represent conflicts and, when applicable, their resolution patterns. The conflict detection and resolution are then formalised as the application of these transformation rules. Note that the approach handles atomic constraints in all the steps of the synchronisation, including normalisation, conflict detection and conflict resolution.

To the best of our knowledge, this work is the first attempt to clarify each step of a work cycle in a centralised approach to optimistic model versioning; i.e., checkout a local copy, make modifications on a local copy, synchronise a local copy, resolve conflicts and commit modifications to a repository. Moreover, this work also constitutes the first attempt to formalise and illustrate constraint-awareness in model versioning.

Specification transformations constitute the basis for normalisation, conflict detection and conflict resolution. In future work, we will analyse termination and confluence in view of DPF. This will facilitate the identification of the conditions under which a set of specification entailments guarantees termination and confluence of the normalisation. Similarly, it will facilitate the identification of the conditions under which a set of conflict resolution patterns guarantees that no new conflicts are introduced.

This chapter further develops the formal approach to model versioning published in [78, 81]. Compared to the previous work, the theoretical foundation and the underlying techniques are extended to handle constraints. Moreover, new examples are added to illustrate how model merging, conflict detection and conflict resolution handle constraints. The findings of this work have already been submitted to a journal for evaluation.

CHAPTER 4

Deep Metamodelling

In this chapter, we present a formal approach to deep metamodelling based on DPF; i.e., a formal approach to metamodelling which supports deep characterisation, double linguistic/ontological typing and linguistic extension.

4.1 Introduction

Domain-specific languages

Models can be specified using general-purpose languages like UML, but to fully unfold the potential of MDE, models are specified using domain-specific languages (DSLs) which are tailored to a specific domain of concern. One way to define DSLs in MDE is by specifying metamodels. In this approach, a system is specified using models at two metalevels: a metamodel defining allowed types and a model instantiating these types. However, this approach may have limitations [7, 10, 42], in particular when the metamodel includes the *type-object* pattern [7, 10, 42], which requires an explicit modelling of types and their instances at the same metalevel. In this case, *deep metamodelling* (also called *multi-level metamodelling*) using more than two metalevels yields simpler models [10].

Deep metamodelling

Deep metamodelling was proposed in the seminal works of Atkinson and Kühne [7], and several researchers and tools have subsequently adopted this approach [5, 6, 27]. However, there is still a lack of formalisation of the main concepts of deep metamodelling such as deep characterisation, double linguistic/ontological typing and linguistic extension. Such formalisation is needed in order to explain the main aspects of the approach, study the different semantic variation points and their consequences, as well as to classify the different semantics found in the tools implementing them [5, 6, 11, 27, 57].

In this chapter, we present a formal approach to deep metamodelling based on DPF; i.e., a formal approach to metamodelling which supports deep characterisation, double linguistic/ontological typing and linguistic extension. The proposed formalisation helps in reasoning about the different semantic variation points in the realisation of deep metamodelling as well as in classifying the existing tools according to these options.

The remainder of the chapter is structured as follows. Section 4.2 illustrates the limitations of traditional metamodelling through an example in the domain of component-based web applications. Section 4.3 introduces deep metamodelling. Section 4.4 explains different concepts of deep metamodelling through its formalisation in DPF. Section 4.5 shows how deep metamodelling relates to traditional metamodelling by means of flattening constructions. In Section 4.6, the current research in deep metamodelling is summarised. In Section 4.7, some concluding remarks and ideas for future work are presented.

4.2 Metamodelling

Traditional metamodelling stack

In a traditional *metamodelling stack* (or hierarchy), models at each metalevel conform to the corresponding metamodel of the modelling language at the adjacent metalevel above (see Figure 4.1). This pattern is often referred to as *linear* metamodelling in the literature [8]. Moreover, in *strict* metamodelling, a model element at each metalevel has exactly one type at the adjacent metalevel above. The top-most model of a traditional metamodelling stack may not conform to any model or may be a reflexive model, i.e., a model which conforms to itself. The length (or depth) of a traditional metamodelling stack is fixed (i.e., it cannot change depending on the requirements) and the metalevels are conventionally numbered from 1 onwards starting from the bottom-most.

OMG's 4-layer hierarchy

For instance, in the 4-layer hierarchy [17] developed by the OMG [66], models conform to the metamodel of UML (see Figure 4.2). The metamodel of UML, in turn, conforms to the metamodel of MOF [68], and the latter is reflexive. Please note that *meta-* is a relative term, so that the UML metamodel is a model as well, while the MOF metamodel is a meta-metamodel with respect to the models.

The OMG's 4-layer hierarchy is the one most widely adopted in practice, but the designer is restricted to working with models at two metalevels only: a metamodel at metalevel M_2 corresponding to the modelling language (e.g., UML or an appropriate DSL), and a model at metalevel M_1 conforming to this metamodel. The following example illustrates that, on some occasions, the restriction to two metalevels leads to the introduction of accidental complexity, which could be avoided if the models were organised using more than two metalevels.

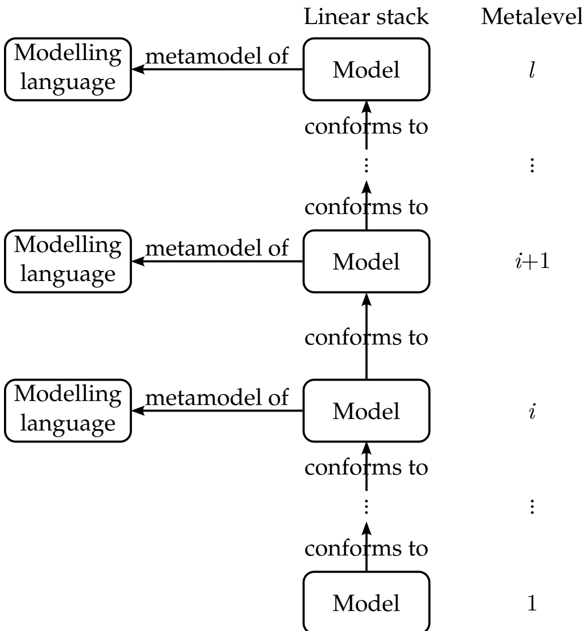


Figure 4.1: Pattern in a linear metamodelling stack

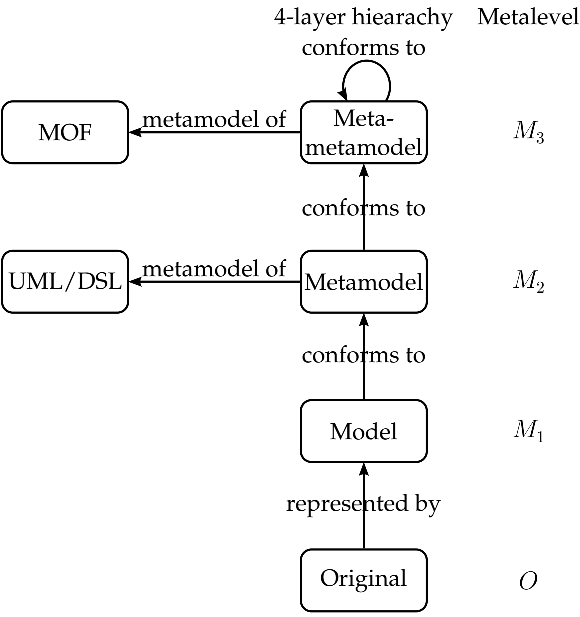


Figure 4.2: OMG's 4-layer hierarchy

Example 26 (A DSL for component-based web applications)**.** *The MeTEOriC project [63] aims at the model-driven engineering of web applications. Here we describe a small excerpt of one of the modelling problems encountered in this project.*

In MeTEOriC, a DSL is adopted to define the mash-up of components (like Google Maps and Google Fusion Tables) to provide the functionality of a web application. A simplified version of this language can be defined using the metalevels M_2 and M_1 of the OMG's 4-layer hierarchy (see Figure 4.3).

The metamodel at metalevel M_2 corresponds to the DSL for component-based web applications. In this metamodel, the metaclass Component *defines component types having a type identifier, whereas the metaclass* CInstance *defines component instances having a variable name and a flag indicating whether it should be visualised. Moreover, the metaassociation* datalink *defines the data link types between component types, whereas the metaassociation* dlinstance *defines the data link instances between component instances. Finally, the metaassociation* type *defines the typing of each component instance.*

The model at metalevel M_1 represents a component-based web application which shows the position of professors' offices on a map. In this model, the classes Map *and* Table *are instances of the metaclass* Component *and represent component types, whereas the classes* UAMCamp *and* UAMProfs *are instances of the metaclass* CInstance *and represent component instances of* Map *and* Table*, respectively. The association* geopos *is an instance of the metaassociation* datalink *and represents the allowed data link between the component types* Map *and* Table*, whereas the association* offices *is an instance of the metaassociation* dlinstance *and represents the actual data link between the component instances* UAMCamp *and* UAMProfs*. Finally, the associations* camptype *and* profstype *are instances of the metaassociation* type *and represent the typing of the component instances* UAMCamp *and* UAMProfs*, respectively.*

The type-object relation between component types and instances is represented explicitly in the metamodel by the metaassociation type *between the metaclasses* Component *and* CInstance*. However, the type-object relation between allowed and actual data links is implicit since there is no explicit relation between the metaassociations* datalink *and* dlinstance*, and this may lead to several problems. Firstly, it is not possible to define that the data link instance* offices *is typed by the data link type* geopos*, which could be particularly ambiguous if the model contained multiple data link types between the component types* Map *and* Table*. Moreover, it could be possible to specify a reflexive data link instance from the component instance* UAMProfs *to itself, which should not be allowed since the component type* Table *does not have any reflexive data link type. Although these errors could be detected by complementing the metamodel with attached OCL constraints, these constraints are not enough to guide the correct instantiation of each data link, in the same way as a built-in type system would do if the data link types and instances belonged to two different metalevels.*

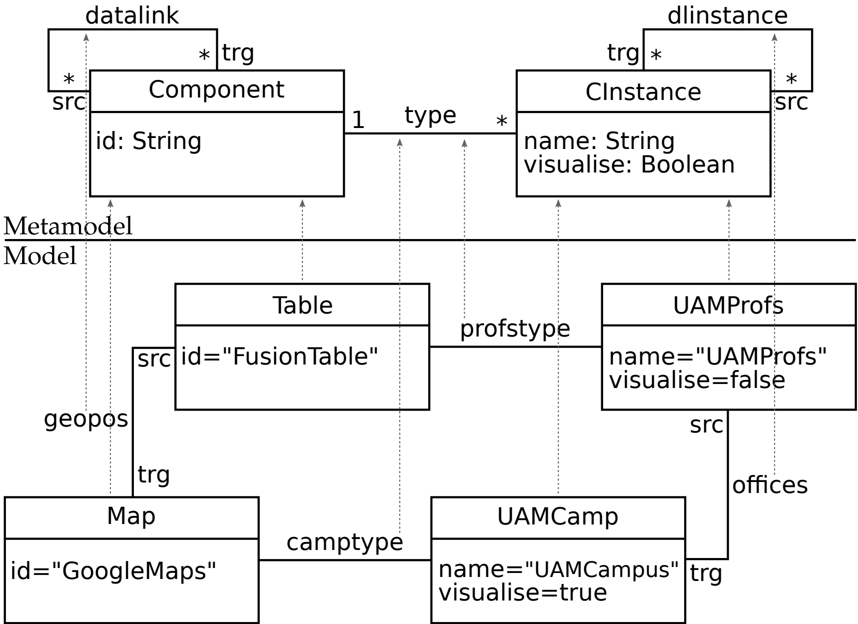


Figure 4.3: A two-metalevel DSL for component-based web applications

In the complete definition of the DSL, the component types can define features which need to be correctly instantiated in the component instances. This leads to even more cluttered models (see Figure 4.4). In the model, the class Scroll *is associated to the class* Map *and represents the zooming capabilities of the map component. The definition of the class* UAMScroll *and its association to both the classes* UAMCamp *as well as* Scroll *has to be done manually. Moreover, the conformance check that the value* true *assigned to the attribute* value *is actually a boolean has to be done manually as well. Hence, either one builds manually the needed machinery to emulate the existence of two metalevels within the same one, or this two-metalevel solution eventually becomes convoluted and hardly usable.*

In the following, we show that organising the models in three metalevels results in a simpler DSL.

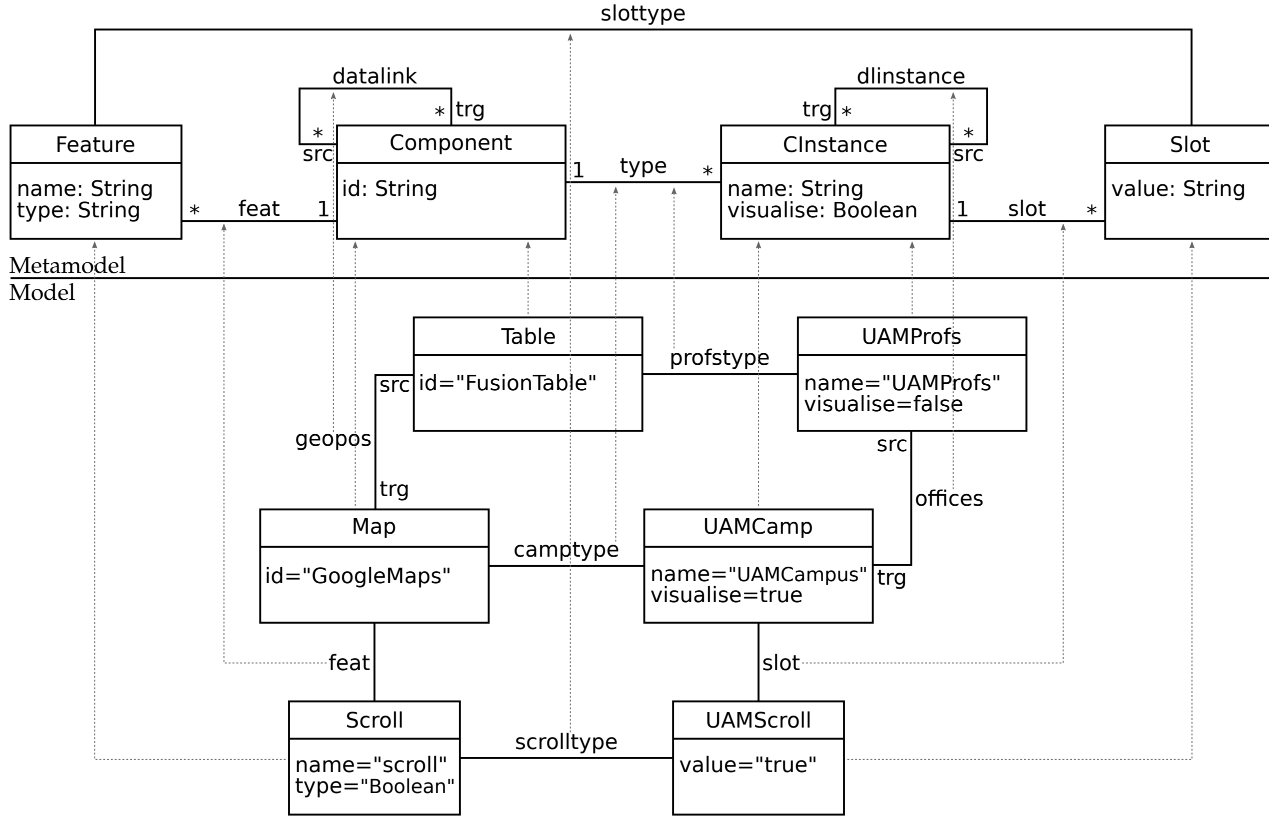


Figure 4.4: Extension of the two-metalevel DSL adding component features

4.3 Deep metamodelling

This section introduces the main concepts of deep metamodelling, illustrating how they overcome the problems of the two-metalevel approach when defining DSLs which incorporate the type-object pattern.

4.3.1 Deep characterisation

The first ingredient of deep metamodelling is *deep characterisation*: the ability to describe structure and express constraints for metalevels below the adjacent one. In this work, we adopt the deep characterisation approach described in [7]. In this approach, each element has a *potency*. In the original proposal of [7], the potency is a natural number which is attached to a model element to describe at how many subsequent metalevels this element can be instantiated. Moreover, the potency decreases in one unit at each instantiation at a deeper metalevel. When it reaches zero, a pure instance that cannot be instantiated further is obtained. In Section 4.4, we provide a more precise definition for potency.

The following example illustrates the usage of deep characterisation. Note that in deep metamodelling, the elements at the top metalevel are pure types, the elements at the bottom metalevel are pure instances, and the elements at intermediate metalevels retain both a type and an instance facet. Because of that, they are all called *clabjects*, which is the merge of the words class and object [10]. Note also that since in deep metamodelling the number of metalevels may change depending on the requirements, we find it more convenient to number the metalevels from 1 onwards starting from the top-most, in contrast to the traditional metamodelling stack (see Figure 4.1).

Example 27 (A DSL for component-based web applications in three metalevels)**.** *Compared to Example 26, the DSL for component-based web applications can be defined in a simpler way using deep metamodelling (see Figure 4.5).*

The model M_1 contains the definition of the DSL. In this model, the clabject Component *has potency* 2*, which denotes that it can be instantiated at the two subsequent metalevels. Its attribute* id *has potency* 1*, which denotes that it can be assigned a value when* Component *is instantiated at the adjacent metalevel below. Its other two attributes* name *and* visualise *have potency* 2*, which denotes that they can be assigned a value only two metalevels below. The association* datalink *also has potency* 2*, which denotes that it can be instantiated at the two subsequent metalevels. The DSL in Figure 4.5 is simpler than the one in Figure 4.3, as it contains less model elements to define the same DSL.*

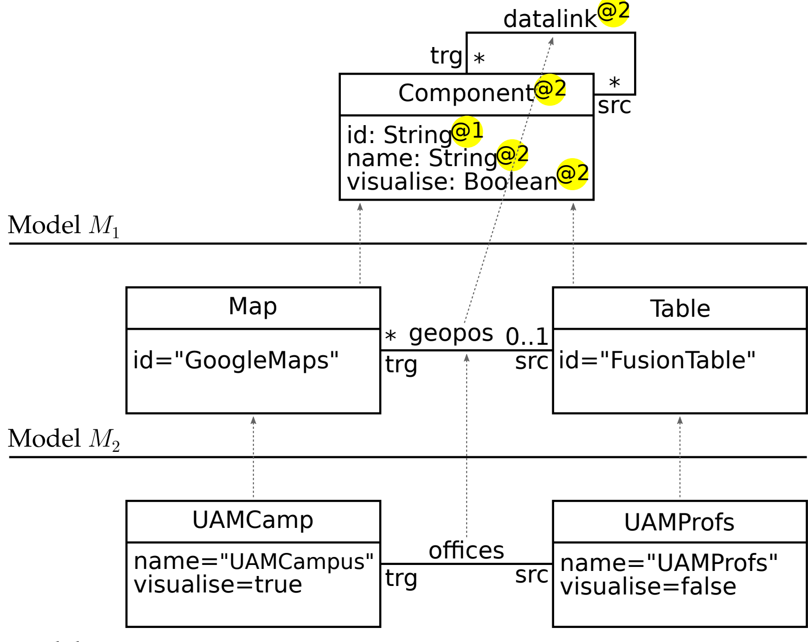


Figure 4.5: A three-metalevel DSL for component-based web applications corresponding to the DSL in Figure 4.3

The deep characterisation is very useful in the design of this DSL. For instance, in the model M_1, the designer can specify the attributes name *and* visualise *which should be assigned a value in indirect instances of* Component*, i.e.,* UAMCamp *and* UAMProfs*. Moreover, the model M_1 does not need to include a clabject* CInstance *or an association* dlinstance *since the clabjects* UAMCamp *and* UAMProfs *are instances of the clabjects* Map *and* Table*, respectively, which in turn are instances of the clabject* Component*.*

4.3.2 Double typing and linguistic extension

The dashed grey arrows in Fig. 4.5 denote the *ontological typing* for the clabjects, as they represent instantiations within a domain; e.g., the clabjects Map and Table are ontologically typed by the clabject Component. In addition, deep metamodelling frameworks usually support an orthogonal *linguistic typing* [10, 27] which refers to the metamodel of the metamodelling language used to specify the models.

Figure 4.6 shows the scheme of this double linguistic/ontological typing. Moreover, it shows a simplified linguistic metamodel, which contains some of the metaclasses needed to specify models, e.g., clabjects, attributes and associations.

Double linguistic/ontological typing

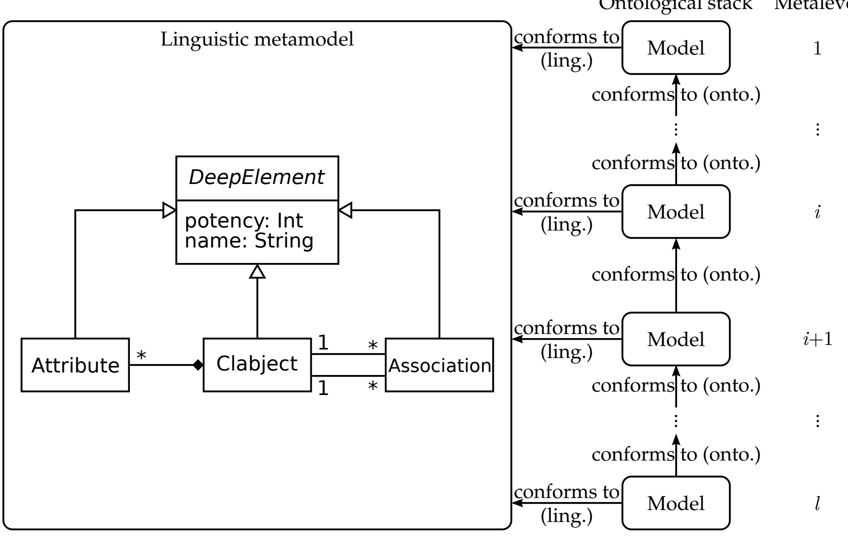


Figure 4.6: Metamodelling stack with double linguistic/ontological typing

In Figure 4.5, the clabjects Component, Map and UAMCamp are linguistically typed by the metaclass Clabject, whereas the attributes id, name and visualise are linguistically typed by the metaclass Attribute. The availability of a double linguistic/ontological typing has the advantage that one can uniformly treat all clabjects independently of their ontological type and metalevel. This enables the specification of *generic* model manipulations typed by the linguistic metamodel, which then become applicable to models at any metalevel.

Linguistic extension

The double linguistic/ontological typing also enables so-called *linguistic extensions* [27]. The crucial observation is that any model in an ontological stack conforms linguistically to the linguistic metamodel. Hence, one can add clabjects which are only linguistically typed, or add new attributes to existing clabjects which are ontologically typed. Figure 4.7 shows the scheme of linguistic extensions. All models in the ontological stack conform linguistically to the linguistic metamodel, but only portions of them conform ontologically to the model at the adjacent metalevel above.

Linguistic extensions are a useful mechanism to design extensible deep DSLs. These extensions are necessary to address new requirements at lower metalevels which could not be foreseen or addressed at the topmost metalevel. The following example illustrates the usage of linguistic extensions.

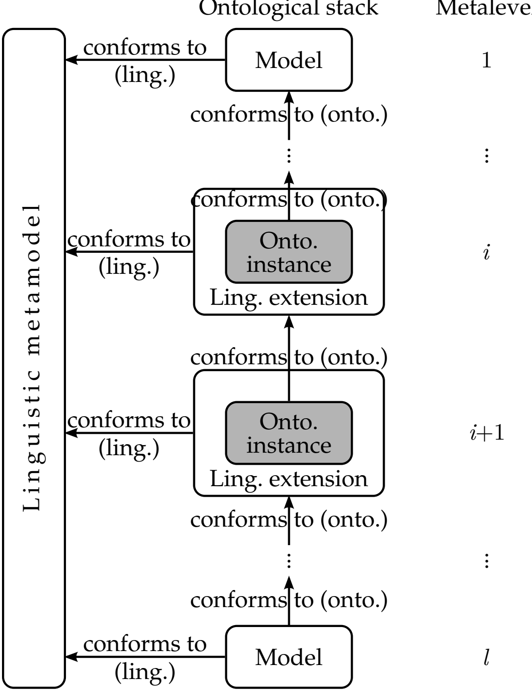


Figure 4.7: Metamodelling stack with double linguistic/ontological typing and linguistic extension

Example 28 (Extended DSL for component-based web applications in three metalevels)**.** *As discussed in Example 26, the component types can define features which need to be correctly instantiated in the component instances. These new features can be naturally expressed as linguistic extensions in the model* M_2 *(see Figure 4.8). In particular, the clabject* Map *is extended with an attribute* scroll *of type* Boolean*. The attribute* scroll *has potency* 1*, which denotes that it can be assigned a value in the model* M_3*.*

Figure 4.8 also shows that potency can be attached to constraints as well. The attached OCL constraint in the model M_1 *forbids to reflexively connect indirect instances of* Component*. This constraint has potency* 2*, which denotes that it has to be evaluated in the model* M_3 *only.*

Regarding the handling of features of component types, the solution presented in Example 28 has two main advantages with respect to the solution in Example 26. Firstly, linguistic extensions enable the use of a built-in type system to check the conformance of feature types and instances; e.g., the conformance check that the value true assigned to the attribute scroll is actually a boolean. Secondly, the built-in type system is used to guide the

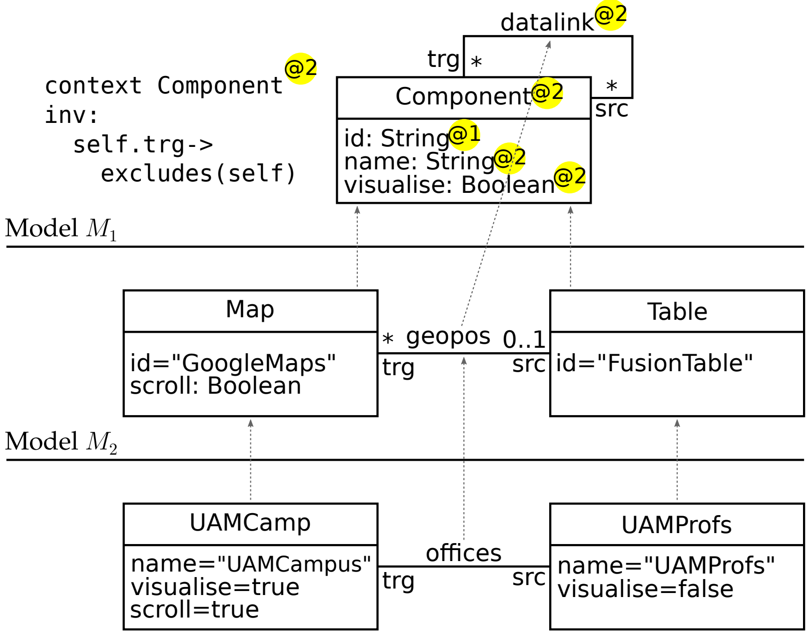


Figure 4.8: Linguistic extension of the three-metalevel DSL adding component features

instantiation of clabjects; e.g., when the clabject Map is instantiated, all its attributes are instantiated as well. In Example 26, the correct instantiation was done either manually or by additional machinery needed to emulate the existence of two metalevels within the same one.

In the following, we discuss some open questions in deep metamodelling.

4.3.3 Some open questions in deep metamodelling

Deep metamodelling allows a more flexible approach to metamodelling by introducing richer modelling mechanisms. However, their semantics have to be precisely defined in order to obtain sound, robust models. Even if the literature (and this section) permits grasping an intuition of how these modelling mechanisms work, there are still open questions which require clarification.

POTENCY ON ASSOCIATIONS

Some works in the literature give different semantics to the potency of associations. In Example 28, the associations are instantiated like clabjects. In this case, the association datalink with potency 2 in the model M_1 is

first instantiated as the association geopos with potency 1 in the model M_2, and then instantiated as the association offices with potency 0 in the model M_3 (see Figure 4.8); i.e., the instantiation of offices is mediated by geopos. In contrast, the attributes name and visualise with potency 2 in the model M_1 are assigned a value directly in the model M_3 (see Figure 4.8); i.e., the instantiation of name and visualise is not mediated. Some frameworks such as EMF [32, 86] represent associations as Java references, so the associations could also be instantiated like attributes. In this case, the association datalink would not need to be instantiated in the model M_2 in order to be able to instantiate it in the model M_3. This would have the effect that one could add an association between any two component instances in the model M_3, not necessarily between instances of Table and instances of Map.

Potency on constraints

Another ambiguity concerns constraints, since some works in the literature support potency on constraints [27] but others do not [11]. In Example 28, the attached OCL constraint in the model M_1 is evaluated in the model M_3 only; i.e., it is not evaluated in the model M_2. In other cases, it might be useful to have a potency which denotes that a constraint has to be evaluated at every metalevel. In Example 28, none of the multiplicity constraints has potency and they are all evaluated at the adjacent metalevel below. In other cases, it might be useful to attach a potency to multiplicity constraints. For instance, a potency 2 on the multiplicity constraints of the association datalink would have the effect that one could control the number of data link instances in the model M_3.

Flattening of deep metamodelling

Finally, another research question concerns the relation between metamodelling stacks with and without deep characterisation. One could define constructions to *flatten* deep characterisation; e.g., given the three-metalevel stack of Example 28, one could obtain another three-metalevel stack without potencies but with some elements replicated along metalevels, making explicit the semantics of potency. This would allow the migration of deeply characterised systems into tools that do not support deep characterisation. One could also define further constructions to flatten multiple metalevels into two or to eliminate the double typing.

Altogether, we observe a lack of consensus and precise semantics for some of the aspects of deep metamodelling. The contribution of this work is the use of DPF to provide a neat semantics for the different aspects of deep metamodelling: double linguistic/ontological typing, linguistic extension and deep characterisation through potency. As a distinguishing note, we propose two possible semantics of potency for each model element, i.e., clabjects, attributes, associations and constraints. To the best of our knowledge, this is the first time that the two semantics have been recognised and formalised.

4.4 Formalisation of deep metamodelling

This section presents a formalisation of deep metamodelling based on DPF. This formalisation is presented stepwise by defining and illustrating double linguistic/ontological conformance, linguistic extension and deep characterisation.

4.4.1 Double metamodelling stack

Recall that in a metamodelling stack which supports double linguistic/ontological conformance – hereafter called *double metamodelling stack* –, models at each metalevel conform linguistically to the corresponding metamodel of a fixed linguistic modelling language and conform ontologically to the model at the adjacent metalevel above (see Section 4.3).

The metamodel of the linguistic modelling language of a deep metamodelling stack can be represented in DPF by a specification $\mathfrak{LM} = (LM, C^{\mathfrak{LM}}: \Sigma)$ which consists of an underlying graph LM and a set of atomic constraints $C^{\mathfrak{LM}}$ specified by means of a predicate signature Σ.

A model at metalevel i of a double metamodelling stack can be represented in DPF by a specification $\mathfrak{S}_i = (S_i, C_i : \Omega)$ which consists of an underlying graph S_i and a set of atomic constraints C_i specified by means of a predicate signature Ω. Moreover, $\mathfrak{S}_i$ conforms linguistically to the specification $\mathfrak{LM}$; i.e., there exists a total linguistic typing morphism $\lambda_i : S_i \to LM$ such that (S_i, λ_i) is a valid instance of $\mathfrak{LM}$. Furthermore, $\mathfrak{S}_i$ conforms ontologically to the specification $\mathfrak{S}_{i-1}$; i.e., there exists a total *two-level* ontological typing morphism $\omega_i : S_i \to S_{i-1}$ such that the ontological typing is compatible with the linguistic typing and (S_i, ω_i) is a valid instance of $\mathfrak{S}_{i-1}$.

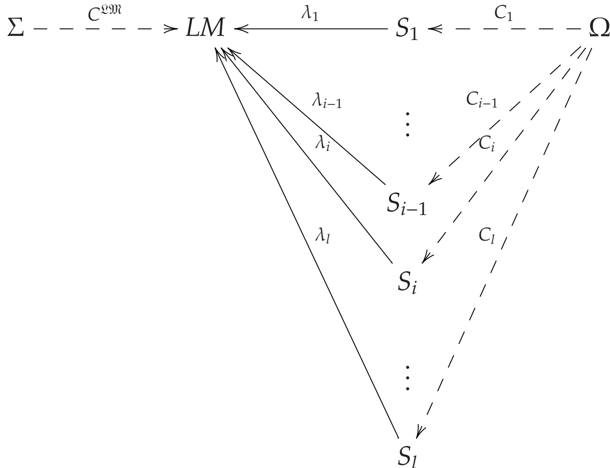
First, in order to enable reuse later in the chapter, the linguistic portion of the double metamodelling stack is defined as follows:

Definition 38 (Linguistic metamodelling stack)**.** *Given:*

- *signatures* $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$, $\Omega = (\Pi^{\Omega}, \alpha^{\Omega})$
- *a specification* $\mathfrak{LM} = (LM, C^{\mathfrak{LM}}: \Sigma)$

A linguistic metamodelling stack with length l consists of:

- *specifications* $\mathfrak{S}_i = (S_i, C_i : \Omega)$*, for all* $1 \leq i \leq l$
- *total linguistic typing morphisms* $\lambda_i : S_i \to LM$*, for all* $1 \leq i \leq l$*, such that:*
  - $(S_i, \lambda_i) \in \mathbf{Inst}(\mathfrak{LM})$



Note that a linguistic metamodelling stack is similar to a traditional linear metamodelling stack with two metalevels, where each specification $\mathfrak{S}_i$ conforms to the specification $\mathfrak{LM}$. Based on this, the double metamodelling stack is constructed by adding ontological typing morphisms $\omega_i : S_i \to S_{i-1}$ to the linguistic metamodelling stack, as follows:

Definition 39 (Double metamodelling stack)**.** *A double metamodelling stack with length l is a linguistic metamodelling stack with length l together with:*

- *total two-level ontological typing morphisms* $\omega_i : S_i \to S_{i-1}$*, for all* $2 \leq i \leq l$*, such that:*
  - $\omega_i ; \lambda_{i-1} = \lambda_i$
  - $(S_i, \omega_i) \in \textbf{Inst}(\mathfrak{S}_{i-1})$

Σ $C^{\mathfrak{LM}}$ LM λ_1 S_1 C_1 Ω
ω_2
λ_{i-1} C_{i-1}
λ_i C_i ω_{i-1}
= S_{i-1}
λ_l ω_i C_l
S_i
ω_{i+1}
ω_l
S_l

The following example illustrates the usage of a double metamodelling stack.

Example 29 (Double metamodelling stack)**.** *Building upon Example 27, Figure 4.9(a) shows the specification* $\mathfrak{LM}$ *and Figures 4.9(b), (c) and (d) show the specifications* $\mathfrak{S}_1$*,* $\mathfrak{S}_2$ *and* $\mathfrak{S}_3$*, respectively, of a double metamodelling stack. Moreover, Figure 4.9 shows the ontological typing morphisms* ω_2 *and* ω_3 *as dashed grey arrows. Tables 4.1 and 4.2 show the signatures* Σ *and* Ω*, respectively.*

The specification $\mathfrak{LM}$ *corresponds to a metamodelling language for object-oriented structural modelling similar to the one in Figure 4.6. The interested reader may consult [80] for details about the semantics of inheritance in DPF.*

Table 4.1: The signature Σ

$\pi \in \Pi^{\Sigma}$	$\alpha^{\Sigma}(\pi)$	Proposed vis.	Semantic interpretation
[irreflexive]	1 (loop a)	[irr] X (loop f)	$\forall x \in X : x \notin f(x)$

Table 4.2: The signature Ω

$\pi \in \Pi^{\Omega}$	$\alpha^{\Omega}(\pi)$	Proposed vis.	Semantic interpretation
[mult(m,n)]	$1 \xrightarrow{a} 2$	X —f [m..n]→ Y	$\forall x \in X : m \leq \lvert f(x) \rvert \leq n$, with $0 \leq m \leq n$ and $n \geq 1$
[irreflexive]	1 (loop a)	[irr] X (loop f)	$\forall x \in X : x \notin f(x)$

The specifications $\mathfrak{S}_1$*,* $\mathfrak{S}_2$ *and* $\mathfrak{S}_3$ *conform linguistically to* $\mathfrak{LM}$*; i.e., there exist linguistic typing morphisms* $\lambda_1 : S_1 \to LM$*,* $\lambda_2 : S_2 \to LM$ *and* $\lambda_3 : S_3 \to LM$ *such that* (S_1, λ_1)*,* (S_2, λ_2) *and* (S_3, λ_3) *are valid instances of* $\mathfrak{LM}$*. The linguistic typing morphisms* λ_1*,* λ_2 *and* λ_3 *are defined as follows:*

λ_1(*Component*) = *Clabject*
λ_1(*datalink*) = *Reference*
λ_1(*id*) = *Attribute*
λ_1(*String*) = *DataType*
λ_2(*Map*) = λ_2(*Table*) = *Clabject*
λ_2(*geopos*) = *Reference*
λ_2(*idMap*) = λ_2(*idTable*) = *Attribute*
λ_2(*"GoogleMaps"*) = λ_2(*"FusionTable"*) = *DataType*
λ_3(*UAMCamp*) = λ_3(*UAMProfs*) = *Clabject*
λ_3(*offices*) = *Reference*

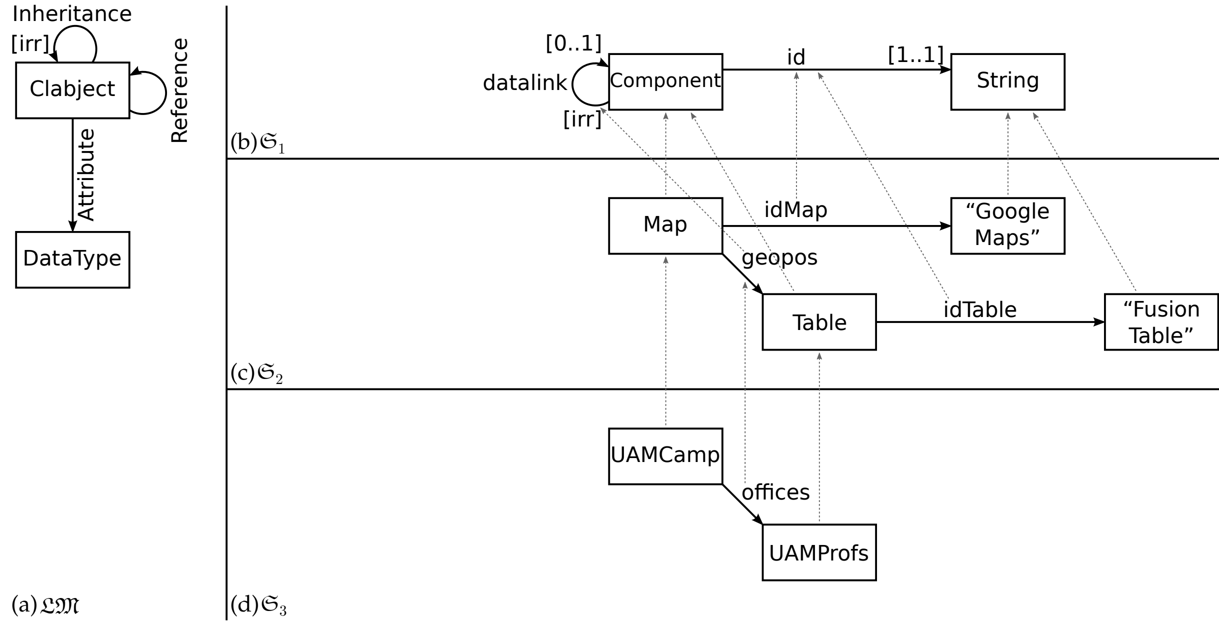


Figure 4.9: The specifications $\mathfrak{LM}$, $\mathfrak{S}_1$, $\mathfrak{S}_2$ and $\mathfrak{S}_3$ together with the ontological typing morphisms ω_2 and ω_3

Moreover, $\mathfrak{S}_2$ *and* $\mathfrak{S}_3$ *conform ontologically to* $\mathfrak{S}_1$ *and* $\mathfrak{S}_2$*, respectively; i.e., there exist total two-level ontological typing morphisms* $\omega_2 : S_2 \to S_1$ *and* $\omega_3 : S_3 \to S_2$ *such that* (S_2, ω_2) *and* (S_3, ω_3) *are valid instances of* $\mathfrak{S}_1$ *and* $\mathfrak{S}_2$*, respectively, and commute with the linguistic typing morphisms. The ontological typing morphisms* ω_2 *and* ω_3 *are defined as follows:*

ω_2(*Map*) = ω_2(*Table*) = *Component*
ω_2(*geopos*) = *datalink*
ω_2(*idMap*) = ω_2(*idTable*) = *id*
ω_2(*"GoogleMaps"*) = ω_2(*"FusionTable"*) = *String*
ω_3(*UAMCamp*) = *Map*
ω_3(*UAMProfs*) = *Table*
ω_3(*offices*) = *geopos*

The proposed double metamodelling stack conveniently represents linguistic and ontological typing, but lacks support for linguistic extension and deep characterisation.

Firstly, in Example 28, the attribute scroll constitutes a linguistic extension of the model at metalevel 2 as this element is only typed linguistically. In Example 29, in contrast, $\mathfrak{S}_2$ can not include an attribute scroll which is not ontologically typed by an element in $\mathfrak{S}_1$. This is because the proposed double metamodelling stack has total ontological typing morphisms rather than partial ones.

Moreover, in Example 28, the deep characterisation of the elements Component and datalink at metalevel 1 forbids that these elements are instantiated at metalevel 4 or below. In Example 29, in contrast, one could add a specification $\mathfrak{S}_4$ including elements that are ontologically typed by elements in $\mathfrak{S}_3$.

Furthermore, in Example 28, the deep characterisation of the attribute name at metalevel 1 allows that this element is instantiated (i.e., it is assigned a value) at metalevel 3. In Example 29, in contrast, $\mathfrak{S}_3$ can not include elements which are ontologically typed by a possible attribute name in $\mathfrak{S}_1$ since $\mathfrak{S}_3$ is ontologically typed by $\mathfrak{S}_2$ but not by $\mathfrak{S}_1$.

Finally, in Example 28, the deep characterisation of the OCL constraint ensures that this constraint is evaluated at metalevel 3. In Example 29, in contrast, the atomic constraint `([irreflexive],` δ_2`)` corresponding to the OCL constraint above is evaluated in $\mathfrak{S}_2$ but not in $\mathfrak{S}_3$. This is because $\mathfrak{S}_2$ conforms ontologically to $\mathfrak{S}_1$, while $\mathfrak{S}_3$ conforms ontologically to $\mathfrak{S}_2$ but not to $\mathfrak{S}_1$.

In the following, we revise the definition of the double metamodelling stack to support linguistic extension as well as different mechanisms of deep characterisation.

4.4.2 Partial double metamodelling stack

Recall that in a metamodelling stack which supports double linguistic/ontological conformance and linguistic extension – hereafter called *partial double metamodelling stack* –, models at each metalevel conform linguistically to the metamodel of a fixed linguistic modelling language, but only a portion of the same models conform ontologically to the model at the adjacent metalevel above (see Section 4.3); i.e., there can be elements in a model which are only linguistically typed.

In analogy to the double metamodelling stack, a model at metalevel i of a partial double metamodelling stack can be represented in DPF by a specification $\mathfrak{S}_i = (S_i, C_i\colon \Omega)$ which conforms linguistically to the specification $\mathfrak{LM}$. In contrast to the double metamodelling stack, however, only a subgraph of $\mathfrak{S}_i$ conforms ontologically to the specification $\mathfrak{S}_{i-1}$; i.e., there exists a partial two-level ontological typing morphism $\omega_i : S_i \multimap S_{i-1}$ which is given by a subgraph $I_i \sqsubseteq S_i$ representing the domain of definition of ω_i (see Definition 47) and a total two-level ontological typing morphisms $\omega_i : I_i \to S_{i-1}$, such that the ontological typing is compatible with the linguistic typing and (I_i, ω_i) is a valid instance of $\mathfrak{S}_{i-1}$.

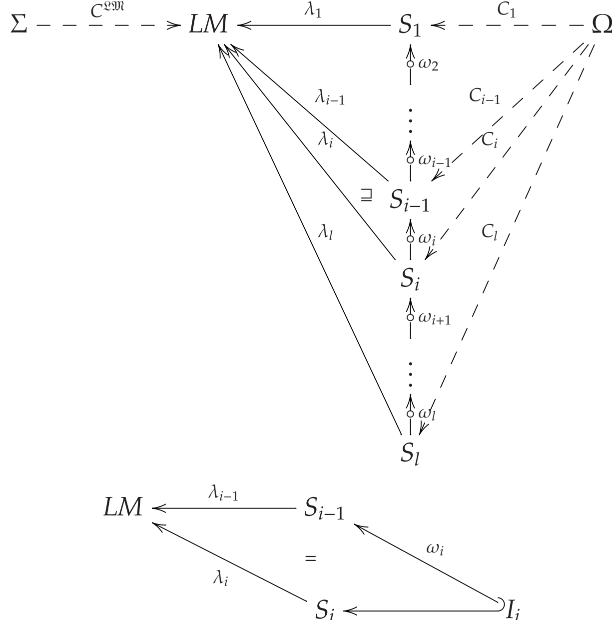
The partial double metamodelling stack is defined as follows:

Definition 40 (Partial double metamodelling stack)**.** *A partial double metamodelling stack with length l is a linguistic metamodelling stack with length l together with:*

- *partial two-level ontological typing morphisms $\omega_i : S_i \multimap S_{i-1}$, for all $2 \le i \le l$, which are given by:*
  - *domain of definition subgraphs $I_i \sqsubseteq S_i$*
  - *total two-level ontological typing morphisms $\omega_i : I_i \to S_{i-1}$*

  such that:

  - $\omega_i ; \lambda_{i-1} \sqsubseteq \lambda_i$
  - $(I_i, \omega_i) \in \mathbf{Inst}(\mathfrak{S}_{i-1})$



Note that partial two-level ontological typing morphisms $\omega_k : S_k \multimap S_{k-1}$ can be composed to obtain a partial *multi-level* ontological typing morphism $\omega_k^i : S_k \multimap S_i$, for all $1 \leq i < k \leq l$, which is given by a subgraph $I_k^i \sqsubseteq S_k$ representing the domain of definition of ω_k^i and a total multi-level ontological typing morphism $\omega_k^i : I_k^i \rightarrow S_i$, where $\omega_k^i = \omega_k; \ldots; \omega_{i-1}$, $I_k^i = (\omega_k^i)^{-1}(S_i) \sqsubseteq I_k$ and $I_k^i \sqsubseteq \ldots \sqsubseteq I_k^{k-1} = I_k$ (see Definition 47).

Example 30 (Partial double metamodelling stack)**.** *Figure 4.10(a) shows the specification $\mathfrak{LM}$ and Figures 4.10(b), (c) and (d) show the specifications $\mathfrak{S}_1$, $\mathfrak{S}_2$ and $\mathfrak{S}_3$, respectively, of a partial double metamodelling stack. Moreover, Figure 4.10 shows the ontological typing morphisms ω_2 and ω_3 as dashed grey arrows.*

Compared to Example 29, the specification $\mathfrak{S}_2$ is extended with an attribute scroll *with data type* Boolean*, while the specification $\mathfrak{S}_3$ is extended with a corresponding data value* true*. The linguistic typing morphisms λ_1, λ_2 and λ_3 are extended with the following mappings:*

λ_2(*scroll*) = *Attribute*
λ_2(*Boolean*) = *DataType*
λ_3(*scrollUAM*) = *Attribute*
λ_3(*true*) = *DataType*

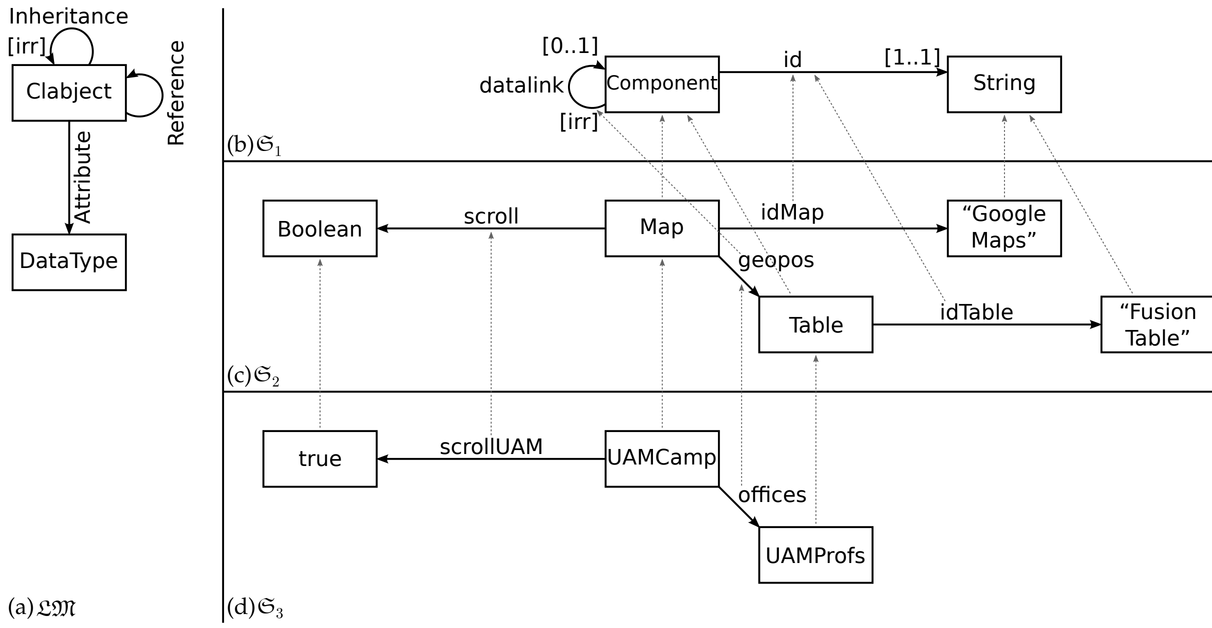


Figure 4.10: The specifications $\mathfrak{LM}$, $\mathfrak{S}_1$, $\mathfrak{S}_2$ and $\mathfrak{S}_3$ together with the ontological typing morphisms ω_2 and ω_3

Moreover, the subgraphs I_2 and I_3 of the specifications $\mathfrak{S}_2$ and $\mathfrak{S}_3$, respectively, conform ontologically to $\mathfrak{S}_1$ and $\mathfrak{S}_2$, respectively; i.e., there exist partial two-level ontological typing morphisms $\omega_2 : S_2 \rightarrow S_1$ and $\omega_3 : S_3 \rightarrow S_2$ such that (I_2, ω_2) and (I_3, ω_3) are valid instances of $\mathfrak{S}_1$ and $\mathfrak{S}_2$, respectively. Note that in this case, the subgraph I_3 is equal to the underlying graph S_3, meaning that the ontological typing morphism ω_3 is actually total. Compared to Example 29, the ontological typing morphism ω_3 is extended with the following mappings:

$\omega_3(\mathit{scrollUAM}) = \mathit{scroll}$
$\omega_3(\mathit{true}) = \mathit{Boolean}$

The proposed partial double metamodelling stack adds support for linguistic extension, but still lacks support for deep characterisation.

In the following, we further revise the definition of the partial double metamodelling stack to support different mechanisms of deep characterisation.

4.4.3 Deep metamodelling stack

Recall that in a metamodelling stack which supports double linguistic/ontological conformance, linguistic extension and deep characterisation – hereafter called *deep metamodelling stack* –, models at each metalevel conform linguistically to the corresponding metamodel of a fixed linguistic modelling language and a portion of the same models conform ontologically to the models at the metalevels above according to the deep characterisation of elements in these models (see Section 4.3).

A mechanism for deep characterisation is potency, for which different interpretations are possible. In this work, two kinds of potency are distinguished, namely *multi-* and *single*-potency, denoted by the symbols ▲p and △p, respectively.

MULTI-POTENCY

A multi-potency ▲p on a clabject/reference at metalevel i denotes that this clabject/reference can be instantiated *at all metalevels from* $i + 1$ *to* $i + p$ (see Figure 4.11), where the instantiation of this clabject/reference has to be *mediated* and the multi-potency has to be *decreased* at each metalevel; e.g., a clabject with multi-potency 0 at metalevel $i + 2$ which is an instance of a clabject with multi-potency 2 at metalevel i must also be an instance of a clabject with multi-potency 1 at metalevel $i + 1$ which in turn is an instance of the considered clabject with multi-potency 2 at metalevel i (see Figures 4.12 and 4.13). Most deep metamodelling approaches assume multi-potency semantics for clabjects [5, 6, 11, 27, 57]. A multi-potency ▲p on an atomic constraint at metalevel i denotes that this constraint is evaluated at all metalevels from $i + 1$ to $i + p$. Finally, attributes only retain either a type or an instance facet but not both; therefore, the multi-potency on attributes can not be considered.

Metalevel	**Clabject**	**Reference**		
⋮	⋮	⋮	⋮	⋮
i	A▲p	A	a▲p	N
$i+1$	B▲p-1	B	b▲p-1	O
⋮	⋮	⋮	⋮	⋮
$i+p-1$	L▲1	L	l▲1	Y
$i+p$	M▲0	M	m▲0	Z
⋮	⋮	⋮	⋮	⋮

Figure 4.11: Intuition on the semantics of multi-potency

Metalevel	Clabject	Reference		
⋮	⋮	⋮	⋮	⋮
i	$A^{\blacktriangle 2}$	A	$a^{\blacktriangle 2}$	N
$i+1$		B		O
$i+2$	$B^{\blacktriangle 0}$	C	$b^{\blacktriangle 0}$	P
⋮	⋮	⋮	⋮	⋮

Figure 4.12: Invalid instantiation: an element with multi-potency 0 at metalevel $i+2$ can not be a direct instance of an element with multi-potency 2 at metalevel i

Metalevel	Clabject	Reference		
⋮	⋮	⋮	⋮	⋮
i	$A^{\blacktriangle 2}$	A	$a^{\blacktriangle 2}$	N
$i+1$	$B^{\blacktriangle 1}$	B	$b^{\blacktriangle 1}$	O
$i+2$	$C^{\blacktriangle 1}$	C	$c^{\blacktriangle 1}$	P
⋮	⋮	⋮	⋮	⋮

Figure 4.13: Invalid instantiation: an element with multi-potency 1 at metalevel $i+2$ can not be an instance of an element with the same multi-potency at metalevel $i+1$

Single-potency

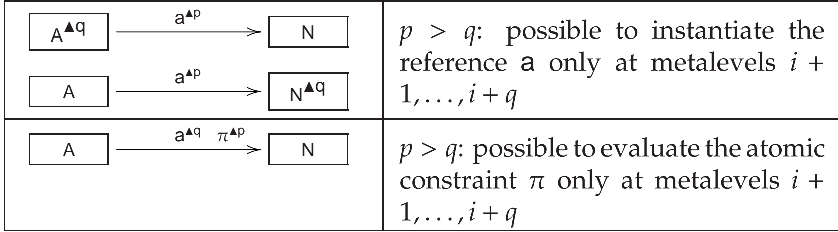
A single-potency $\vartriangle$p on a clabject/reference at metalevel i, in contrast, denotes that this clabject/reference can be instantiated *at metalevel* $i+p$ *only* (see Figure 4.14). A single-potency $\vartriangle$p on an attribute at metalevel i denotes that this attribute can be instantiated (i.e., can be assigned a value) at metalevel $i+p$ only. A single-potency $\vartriangle$p on an atomic constraint at metalevel i denotes that this atomic constraint is evaluated at metalevel $i+p$ only.

Metalevel	Clabject	Reference	Attribute
⋮	⋮	⋮ ⋮ ⋮	⋮ ⋮ ⋮
i	A$^{\vartriangle p}$	A —a$^{\vartriangle p}$→ N	A —a$^{\vartriangle p}$→ DT
⋮	⋮	⋮ ⋮ ⋮	⋮ ⋮ ⋮
$i+p$	B$^{\vartriangle 0}$	M —b$^{\vartriangle 0}$→ Z	M —b$^{\vartriangle 0}$→ DV
⋮	⋮	⋮ ⋮ ⋮	⋮ ⋮ ⋮

Figure 4.14: Intuition on the semantics of single-potency

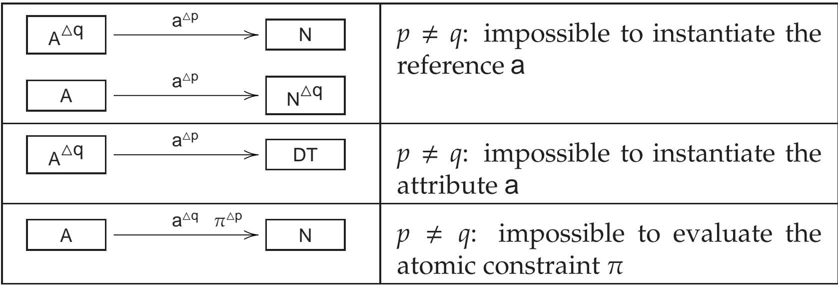
Each element in a model has either a multi-potency or a single-potency. However, some combinations of potencies on interdependent elements may lead to contradictions. Tables 4.3, 4.4 and 4.5 show the contradictory combinations of multi- and single-potencies.

Table 4.3: Contradictory combinations of multi-potencies on interdependent elements



Combination	Contradiction
A$^{\blacktriangle q}$ —a$^{\blacktriangle p}$→ N A —a$^{\blacktriangle p}$→ N$^{\blacktriangle q}$	$p > q$: possible to instantiate the reference a only at metalevels $i+1,\ldots,i+q$
A —a$^{\blacktriangle q}$ $\pi^{\blacktriangle p}$→ N	$p > q$: possible to evaluate the atomic constraint π only at metalevels $i+1,\ldots,i+q$

Table 4.4: Contradictory combinations of single-potencies on interdependent elements



Combination	Consequence
$A^{\triangle q} \xrightarrow{a^{\triangle p}} N$ $A \xrightarrow{a^{\triangle p}} N^{\triangle q}$	$p \neq q$: impossible to instantiate the reference a
$A^{\triangle q} \xrightarrow{a^{\triangle p}} DT$	$p \neq q$: impossible to instantiate the attribute a
$A \xrightarrow{a^{\triangle q} \quad \pi^{\triangle p}} N$	$p \neq q$: impossible to evaluate the atomic constraint π

Table 4.5: Contradictory combinations of multi- and single-potencies on interdependent elements

Combination	Consequence
$A^{\triangle q} \xrightarrow{a^{\blacktriangle p}} N$ $A \xrightarrow{a^{\blacktriangle p}} N^{\triangle q}$	$p \neq q$: impossible to instantiate the reference a $p = q$: possible to instantiate the reference a only if $p = q = 1$
$A^{\blacktriangle q} \xrightarrow{a^{\triangle p}} N$ $A \xrightarrow{a^{\triangle p}} N^{\blacktriangle q}$	$p > q$: impossible to instantiate the reference a
$A^{\blacktriangle q} \xrightarrow{a^{\triangle p}} DT$	$p > q$: impossible to instantiate the attribute a
$A \xrightarrow{a^{\triangle q} \quad \pi^{\blacktriangle p}} N$	$p \neq q$: impossible to evaluate the atomic constraint π $p = q$: possible to evaluate the atomic constraint π only if $p = q = 1$
$A \xrightarrow{a^{\blacktriangle q} \quad \pi^{\triangle p}} N$	$p > q$: impossible to evaluate the atomic constraint π

In analogy to the partial double metamodelling stack, a model at metalevel i of a deep metamodelling stack can be represented in DPF by a specification $\mathfrak{S}_i = (S_i, C_i{:}\Omega)$ which conforms linguistically to the specification $\mathfrak{LM}$. In contrast to the partial double metamodelling stack, however, the specification $\mathfrak{S}_i$ supports deep characterisation; i.e., it is compliant with the following requirements, for all $1 \leq i < j < k \leq l$, with $o = j - i$ and $p = k - i$:

1. Elements in specifications from $\mathfrak{S}_{i+1}$ to $\mathfrak{S}_k$ can be ontologically typed by elements with multi-potency p in a specification $\mathfrak{S}_i$.
2. Elements in a specification $\mathfrak{S}_k$ can be ontologically typed by elements with single-potency p in a specification $\mathfrak{S}_i$.
3. Elements in specifications from $\mathfrak{S}_{i+1}$ to $\mathfrak{S}_k$ satisfy the atomic constraints with multi-potency p in a specification $\mathfrak{S}_i$.
4. Elements in a specification $\mathfrak{S}_k$ satisfy the atomic constraints with single-potency p in a specification $\mathfrak{S}_i$.

The multi- and single-potency of each clabject, reference and attribute in a specification $\mathfrak{S}_i$ can be represented by considering *type-facet subgraphs* $T_i^k \sqsubseteq S_i$ (see Figure 4.15). Elements with multi-potency p in a specification $\mathfrak{S}_i$ are included in the type-facet subgraphs from T_i^{i+1} to T_i^k only. Similarly, elements with single-potency p in a specification $\mathfrak{S}_i$ are included in the type-facet subgraph T_i^k only.

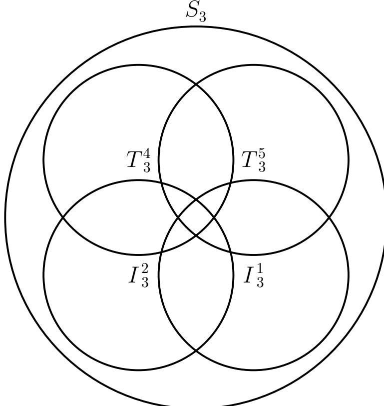


Figure 4.15: A venn diagram illustrating the partitioning of the underlying graph S_3 of a specification $\mathfrak{S}_3$ into possible type-facet and instance-facet subgraphs

Similarly, the multi- and single-potency of each atomic constraint in a specification $\mathfrak{S}_i$ can be represented by considering *subsets of atomic constraints* $C_i^k \sqsubseteq C_i$. Atomic constraints with multi-potency p in a specification $\mathfrak{S}_i$ are included in the subsets from C_i^{i+1} to C_i^k only. Similarly, atomic constraints with single-potency p in a specification $\mathfrak{S}_i$ are included in the subset C_i^k only.

The instantiation in a specification $\mathfrak{S}_k$ of elements with multi- and single-potency p in a specification $\mathfrak{S}_i$ can be represented by considering partial multi-level ontological typing morphisms $\omega_k^i : S_k \multimapdotbothvert S_i$, which are given by *instance-facet subgraphs* $I_k^i \sqsubseteq S_k$ (see Figure 4.15) together with total multi-level ontological typing morphisms $\omega_k^i : I_k^i \rightarrow S_i$ (see Figure 4.16).

The partitioning of a specification into possibly overlapping type-facet subgraphs and instance-facet subgraphs follows the rationale behind the term clabject, namely that elements in a specification of a deep metamodelling stack can retain both a type (class) and instance (object) facet.

Figure 4.16: Partial multi-level ontological typing morphisms

Note that since the instantiation of elements with single-potency can jump over several metalevels, the multi-level ontological typing morphisms ω_k^i and their domains of definition I_k^i can not be obtained by com-

posing the two-level ontological typing morphisms as was the case for partial double metamodelling stacks; they have to be defined explicitly. Moreover, these jumps mean that the instantiation is no longer monotonic, i.e., $I_k^i \sqsubseteq \ldots \sqsubseteq I_k^{k-1}$ does not hold.

The requirements 1 and 2 that all the elements in a specification $\mathfrak{S}_k$ that are ontologically typed by elements in a specification $\mathfrak{S}_i$ actually have to be ontologically typed by elements in the type-facet subgraph T_i^k can be represented by the condition $(\omega_k^i)^{-1}(T_i^k) = I_k^i$.

The requirements 3 and 4 that all the elements in a specification $\mathfrak{S}_k$ that are ontologically typed by elements in the type-facet subgraph T_i^k also have to satisfy the atomic constraints in the subset C_i^k can be represented by the condition that (I_k^i, ω_k^i) is a valid instance of the *type-facet subspecification* $\mathfrak{T}_i^k = (T_i^k, C_i^k{:}\Omega) \sqsubseteq \mathfrak{S}_i$.

The partitioning of a specification into type-facet subspecifications ensures that only valid combinations of potencies are allowed. This is because the contradictory combinations of potencies presented in Table 4.3, 4.4 and 4.5 would lead to dangling arrows or dangling atomic constraints and hence to invalid type-facet subspecifications.

The requirements above, however, are not sufficient to represent all the aspects of the semantics of deep characterisation. A specification $\mathfrak{S}_i$ of a deep metamodelling stack has to be compliant with the following additional requirements:

5. Elements in specifications from $\mathfrak{S}_{k+1}$ to $\mathfrak{S}_l$ can *not* be ontologically typed by elements with multi-potency p in a specification $\mathfrak{S}_i$; i.e., the instantiation of elements with multi-potencies stops when the multi-potency is zero.

6. Elements in specifications from $\mathfrak{S}_{i+1}$ to $\mathfrak{S}_{k-1}$ and from $\mathfrak{S}_{k+1}$ to $\mathfrak{S}_l$ can *not* be ontologically typed by elements with single-potency p in a specification $\mathfrak{S}_i$.

The multi- and single-potency of each clabject, reference and attribute in a specification $\mathfrak{S}_i$ can be distinguished by considering additional *multi-potency subgraphs* $MP_i^k \sqsubseteq T_i^k$ and *single-potency subparts* $SP_i^k = (T_i^k \backslash MP_i^k) \sqsubseteq T_i^k$ (see Figure 4.17).

The requirements 5 can be represented by the condition $(\omega_k^i)^{-1}(MP_i^k \setminus MP_i^{k+1}) \sqsubseteq S_k \setminus (\bigcup_{k'>k} T_k^{k'})$ where $S_k \setminus (\bigcup_{k'>k} T_k^{k'})$ includes all the elements in $\mathfrak{S}_k$ which do not retain a type-facet; i.e., which are not instantiated at any metalevel.

The requirement 6 can be represented by the conditions $(\omega_j^i)^{-1}(SP_i^k) = \emptyset$ and $(\omega_k^i)^{-1}(SP_i^k) \sqsubseteq S_k \setminus (\bigcup_{k'>k} T_k^{k'})$.

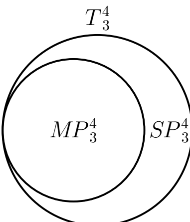


Figure 4.17: A venn diagram illustrating the partitioning of the type-facet subgraph T_3^4 of a specification $\mathfrak{S}_3$ into the multi-potency subgraph MP_3^4 and the single-potency subpart SP_3^4, respectively

Furthermore, a specification $\mathfrak{S}_i$ of a deep metamodelling stack has to be compliant with the following additional requirements:

7. Elements in a specification $\mathfrak{S}_k$ which are ontologically typed by elements with multi-potency p in a specification $\mathfrak{S}_i$ must also be ontologically typed by elements with multi-potency $o < p$ in a specification $\mathfrak{S}_j$ which in turn are ontologically typed by the considered elements with multi-potency p in the specification $\mathfrak{S}_i$; i.e., the instantiation of elements with multi-potency is mediated.

8. Elements with multi-potency q in a specification $\mathfrak{S}_k$ can not be ontologically typed by elements with multi-potency $p \leq q$ in a specification $\mathfrak{S}_i$; i.e., the multi-potency of elements is decreased at each instantiation.

The requirement 7 can be represented by the conditions $MP_i^k \sqsubseteq \ldots \sqsubseteq MP_i^{i+1}$, $\omega_k^j; \omega_j^i \sqsubseteq \omega_k^i$ (i.e., $(\omega_k^j)^{-1}(I_j^i) \sqsubseteq I_k^i$) and $(\omega_k^i)^{-1}(MP_i^k) \sqsubseteq I_k^j$.

The requirement 8 can be represented by the condition $(\omega_j^i)^{-1}(MP_i^k \setminus MP_i^{k+1}) \sqsubseteq (MP_j^k \setminus MP_j^{k+1})$.

Finally, a specification $\mathfrak{S}_i$ of a deep metamodelling stack has to be compliant with the following additional requirements:

9. Elements in a specification have either a multi-potency or a single-potency, but not both.

10. The ontological typing is compatible with the linguistic typing.

The requirement 9 can be represented by the condition $SP_i^j \cap T_i^k = \emptyset$.

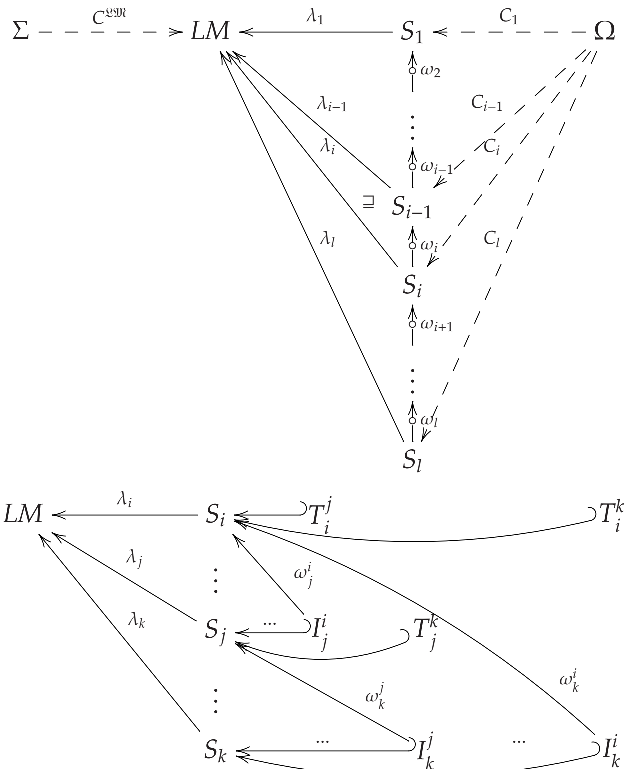
The requirement 10 can be represented as usual by the condition $\omega_k^i; \lambda_i \sqsubseteq \lambda_k$.

Taking into account all these conditions, the deep metamodelling stack is defined as follows:

Definition 41 (Deep metamodelling stack)**.** *A deep metamodelling stack with length l is a linguistic metamodelling stack with length l together with:*

- *type-facet subspecifications* $\mathfrak{T}_i^k = (T_i^k, C_i^k{:}\Omega) \sqsubseteq \mathfrak{S}_i$*, for all* $1 \leq i < k \leq l$
- *multi-potency subgraphs* $MP_i^k \sqsubseteq T_i^k$*, for all* $1 \leq i < k \leq l$*, such that:*
  - $MP_i^k \sqsubseteq \ldots \sqsubseteq MP_i^{i+1}$ *(requirement 7)*
- *single-potency subparts* $SP_i^k = (T_i^k \setminus MP_i^k) \sqsubseteq T_i^k$*, for all* $1 \leq i < k \leq l$*, such that:*
  - $SP_i^j \cap T_i^k = \emptyset$*, for all* $j \neq k$ *(requirement 9)*
- *partial multi-level ontological typing morphisms* $\omega_k^i : S_k \multimap S_i$ *, for all* $1 \leq i < k \leq l$*, which are given by:*
  - *instance-facet subgraphs* $I_k^i \sqsubseteq S_k$
  - *total multi-level ontological typing morphisms* $\omega_k^i : I_k^i \rightarrow S_i$

  such that for all $1 \leq i < k \leq l$ *and all* $i < j < k$*:*
  - $(\omega_k^i)^{-1}(T_i^k) = I_k^i$ *(requirements 1 and 2)*
  - $(I_k^i, \omega_k^i) \in \mathbf{Inst}(\mathfrak{T}_i^k)$ *(requirements 3 and 4)*
  - $(\omega_k^i)^{-1}(MP_i^k \setminus MP_i^{k+1}) \sqsubseteq S_k \setminus (\bigcup_{k'>k} T_k^{k'})$ *(requirement 5)*
  - $(\omega_j^i)^{-1}(SP_i^k) = \emptyset$ *(requirement 6)*
  - $(\omega_k^i)^{-1}(SP_i^k) \sqsubseteq S_k \setminus (\bigcup_{k'>k} T_k^{k'})$ *(requirement 6)*
  - $\omega_k^j; \omega_j^i \sqsubseteq \omega_k^i$ *(i.e.,* $(\omega_k^j)^{-1}(I_j^i) \sqsubseteq I_k^i$*)* *(requirement 7)*
  - $(\omega_k^i)^{-1}(MP_i^k) \sqsubseteq I_k^j$ *(requirement 7)*
  - $(\omega_j^i)^{-1}(MP_i^k \setminus MP_i^{k+1}) \sqsubseteq (MP_j^k \setminus MP_j^{k+1})$ *(requirement 8)*
  - $\omega_k^i; \lambda_i \sqsubseteq \lambda_k$ *(requirement 10)*



Example 31 (Deep metamodelling stack)**.** *Building upon Example 30, Figure 4.18(a) shows the specification $\mathfrak{LM}$ and Figures 4.18(b), (c) and (d) show the specifications $\mathfrak{S}_1$, $\mathfrak{S}_2$ and $\mathfrak{S}_3$. Moreover, Figure 4.18 shows the ontological typing morphisms ω_2^1 and ω_3^2 as dashed grey arrows. Figure 4.19 shows the same specifications and the ontological typing morphism ω_3^1.*

In analogy to Example 30, $\mathfrak{S}_1$, $\mathfrak{S}_2$ and $\mathfrak{S}_3$ conform linguistically to $\mathfrak{LM}$.

In contrast to Example 30, however, the multi-potency ▲2 on the clabject Component *and the reference* datalink *denotes that these elements are in both type-facet subgraphs T_1^2 and T_1^3 (as well as the multi-potency subgraphs MP_1^2 and MP_1^2). Moreover, the single-potency △1 on the attribute* id *denotes that this element is in the type-facet subgraph T_1^2 only (as well as the single-potency subpart SP_1^2), while the single-potency △1 on the atomic constraint (`[mult(1,1)]`, δ_3) on the same attribute denotes that this element is in the subset of atomic constraints C_1^2 only. Furthermore, the single-potency △2 on the attribute* name *denotes that this element is in the type-facet subgraph T_1^3 only (as well as the single-potency subpart SP_1^3), while the single-potency △2 on the atomic constraint (`[mult(1,1)]`, δ_4) on the same attribute denotes that this element is in the subset of atomic constraints C_1^3 only.*

The specification $\mathfrak{S}_2$ *conforms ontologically to* $\mathfrak{S}_1$*; i.e., there exists a partial multi-level ontological typing morphism* $\omega_2^1 : S_2 \multimap S_1$ *such that* (I_2^1, ω_2^1) *is a valid instance of the type-facet subspecification* $\mathfrak{T}_1^2 = (T_1^2, C_1^2 : \Omega)$*. The ontological typing morphism* ω_2^1 *is defined as follows (see Figure 4.18):*

ω_2^1(*Map*) = ω_2^1(*Table*) = *Component*
ω_2^1(*geopos*) = *datalink*
ω_2^1(*idMap*) = ω_2^1(*idTable*) = *id*
ω_2^1(*"GoogleMaps"*) = ω_2^1(*"FusionTable"*) = *String*

The specification $\mathfrak{S}_3$ *conforms ontologically to both* $\mathfrak{S}_2$ *and* $\mathfrak{S}_1$*; i.e., there exists partial multi-level ontological typing morphisms* $\omega_3^2 : S_3 \multimap S_2$ *and* $\omega_3^1 : S_3 \multimap S_1$ *such that* (I_3^2, ω_3^2) *and* (I_3^1, ω_3^1) *are valid instances of the type-facet subspecifications* $\mathfrak{T}_2^3 = (T_2^3, C_2^3 : \Omega)$ *and* $\mathfrak{T}_1^3 = (T_1^3, C_1^3 : \Omega)$*, respectively. The ontological typing morphisms* ω_3^2 *and* ω_3^1 *are defined as follows (see Figures 4.18 and 4.19):*

ω_3^2(*UAMCamp*) = *Map*
ω_3^2(*UAMProfs*) = *Table*
ω_3^2(*offices*) = *geopos*
ω_3^2(*scrollUAM*) = *scroll*
ω_3^2(*true*) = *Boolean*
ω_3^1(*UAMCamp*) = ω_3^1(*UAMProfs*) = *Component*
ω_3^1(*offices*) = *datalink*
ω_3^1(*nameMapUAM*) = ω_3^1(*nameTableUAM*) = *name*
ω_3^1(*"UAMCampus"*) = ω_3^1(*"UAMProfs"*) = *String*

It is straightforward to show that this sample deep metamodelling stack satisfies all the conditions in Definition 41.

In this section, we presented a formalisation of deep metamodelling based on DPF from a structural point of view.

In the following, we switch to an operational point of view and show how to flatten deep characterisation by transforming a deep metamodelling stack into a partial double metamodelling stack.

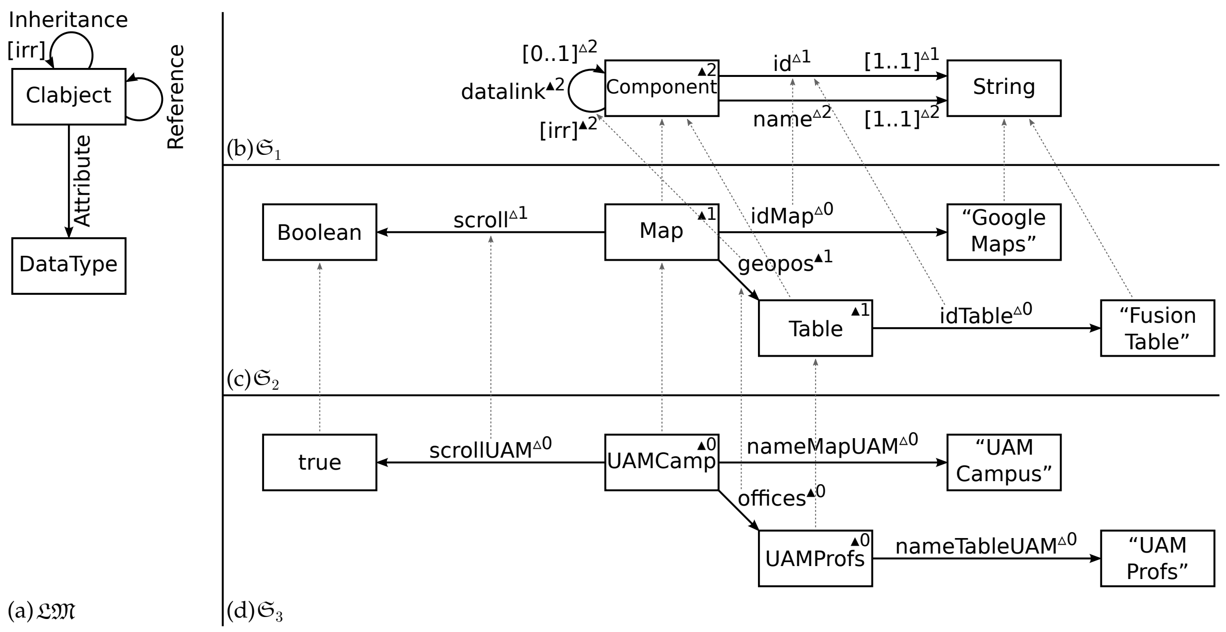


Figure 4.18: The specifications $\mathfrak{LM}$, $\mathfrak{S}_1$, $\mathfrak{S}_2$ and $\mathfrak{S}_3$ together with the ontological typing morphisms ω_2^1 and ω_3^2

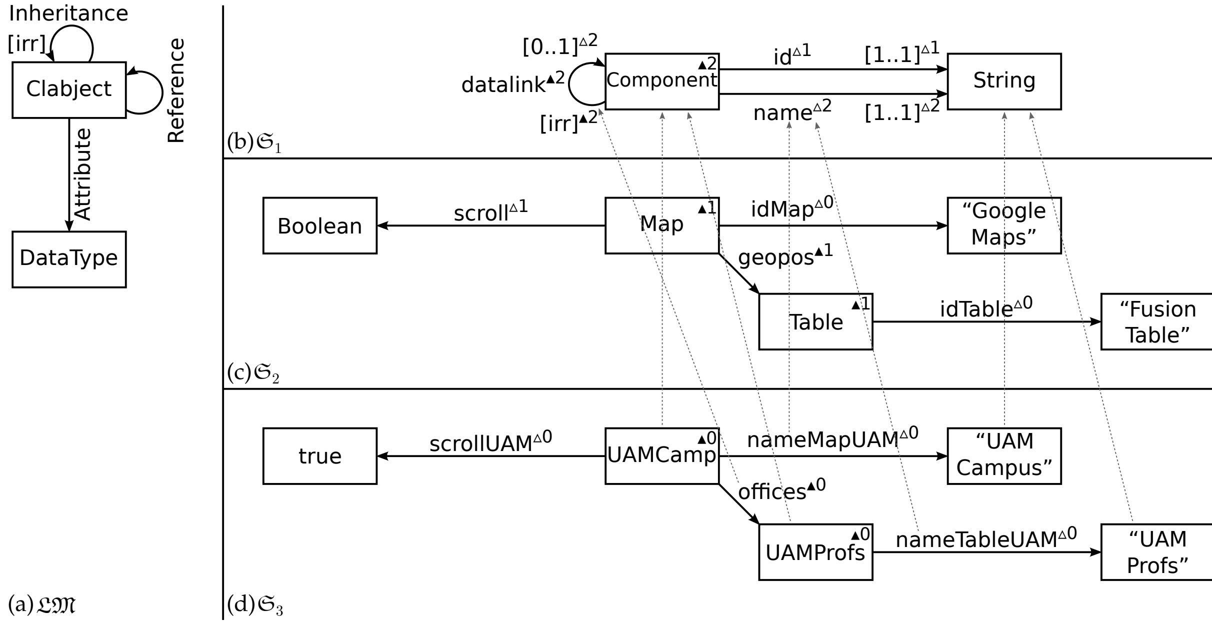


Figure 4.19: The specifications $\mathfrak{LM}$, $\mathfrak{S}_1$, $\mathfrak{S}_2$ and $\mathfrak{S}_3$ together with the ontological typing morphism ω_3^1

4.5 Flattening of a deep metamodelling stack

Recall that in a deep metamodelling stack, an element with single-potency 0 at metalevel k may be ontologically typed by an element with single-potency $p = k - i$ at metalevel i; i.e., there may be p metalevels between an instance and its type. In a double metamodelling stack, in contrast, an element at metalevel k can only be ontologically typed by an element at metalevel $k - 1$. In order to better illustrate the semantics of deep characterisation, we show how to flatten deep characterisation by transforming a deep metamodelling stack into a partial double metamodelling stack. This flattening is defined by multiple *replication rules* and an *extraction rule*.

The replication rules rc_0, rr_1, ra_1 and rac_2 follow a general pattern which, for each element with single-potency $p \geq 2$ at metalevel i, adds to metalevel $k - 1$ a replica of the considered element with single-potency decreased to 1. Similar to the layering of transformation rules in specification transformation (see Section 2.5), the subscripts from 0 to 2 denote the layer to which a rule belongs, so that rules of layer 0 are applied before rules of layer 1, etc.

The replication rule rc_0 adds to metalevel $k - 1$ a replica with single-potency 1 of a clabject with single-potency p at metalevel i, as follows[1]:

Definition 42 (Replication rule rc_0 for clabjects)**.** *Given a deep metamodelling stack with length l, for all $1 \leq i < k \leq l$ and $k \geq i + 2$:*

- *for each* $\mathsf{A} \in SP_i^k$
  - $T'^{k}_{k-1} = T^{k}_{k-1} \cup \mathsf{A}'$
  - $I'^{i}_{k-1} = I^{i}_{k-1} \cup \mathsf{A}'$ *and* $\omega^{i}_{k-1}(\mathsf{A}') = \mathsf{A}$
- *for each* $\mathsf{B} \in I^{i}_{k}$ *such that* $\omega^{i}_{k}(\mathsf{B}) = \mathsf{A}$
  - $I'^{k-1}_{k} = I^{k-1}_{k} \cup \mathsf{B}$ *and* $\omega^{k-1}_{k}(\mathsf{B}) = \mathsf{A}'$

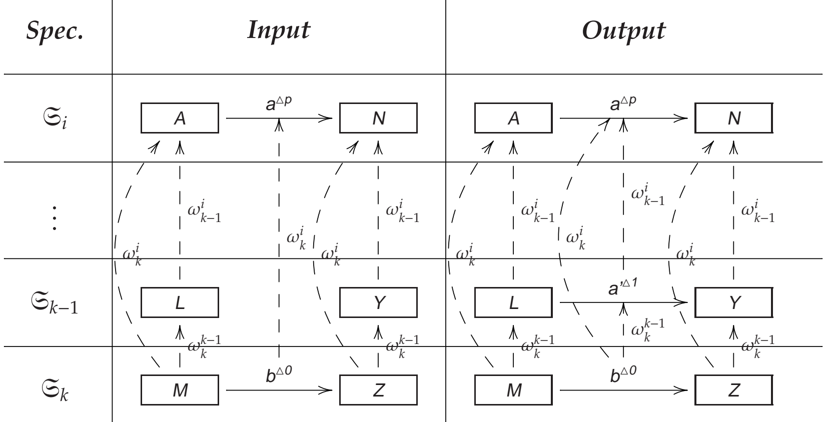
[1] T'^{k}_{k-1} and I'^{i}_{k-1} denote the state of the type- and instance-facet subgraphs T^{k}_{k-1} and I^{i}_{k-1}, respectively, after the application of the rule.

Spec.	*Input*	*Output*
$\mathfrak{S}_i$	$A^{\triangle p}$	$A^{\triangle p}$
$\vdots$		ω^i_{k-1}
	ω^i_k	ω^i_k
$\mathfrak{S}_{k-1}$		$A'^{\triangle 1}$
		ω^{k-1}_k
$\mathfrak{S}_k$	$B^{\triangle 0}$	$B^{\triangle 0}$

The replication rule rr_1 adds to metalevel $k-1$ a replica with single-potency 1 of a reference with single-potency p at metalevel i, as follows:

Definition 43 (Replication rule rr_1 for references)**.** *Given a deep metamodelling stack with length l, for all $1 \leq i < k \leq l$ and $k \geq i+2$:*

- *for each* $(A \xrightarrow{a} N) \in SP^k_i$
  - *for each* $L, Y \in I^i_{k-1}$ *such that* $\omega^i_{k-1}(L) = A$ *and* $\omega^i_{k-1}(Y) = N$
    - $T'^k_{k-1} = T^k_{k-1} \cup (L \xrightarrow{a'} Y)$
    - $I'^i_{k-1} = I^i_{k-1} \cup (L \xrightarrow{a'} Y)$ *and* $\omega^i_{k-1}(L \xrightarrow{a'} Y) = (A \xrightarrow{a} N)$
- *for each* $(M \xrightarrow{b} Z) \in I^i_k$ *such that* $\omega^i_k(M \xrightarrow{b} Z) = (A \xrightarrow{a} N)$, $\omega^{k-1}_k(M) = L$ *and* $\omega^{k-1}_k(Z) = Y$
  - $I'^{k-1}_k = I^{k-1}_k \cup (M \xrightarrow{b} Z)$ *and* $\omega^{k-1}_k(M \xrightarrow{b} Z) = (L \xrightarrow{a'} Y)$



Remark 10 (Identity of data types). *Recall that, similar to E-graphs [35, 36], attributes of nodes can be represented in DPF by arrows from these nodes to nodes representing data types. Nodes representing data types can be regarded as having a "global identity" in a deep metamodelling stack. Therefore, we assume that all nodes representing data types are implicitly available in each specification $\mathfrak{S}_i$ of the deep metamodelling stack.*

The replication rule ra_1 adds to metalevel $k-1$ a replica with single-potency 1 of an attribute with single-potency p at metalevel i, as follows:

Definition 44 (Replication rule ra_1 for attributes). *Given a deep metamodelling stack with length l, for all $1 \leq i < k \leq l$ and $k \geq i+2$:*

- *for each* $(A \xrightarrow{a} DT) \in SP^k_i$
  - *for each* $L, DT \in I^i_{k-1}$ *such that* $\omega^i_{k-1}(L) = A$
    - $T'^k_{k-1} = T^k_{k-1} \cup (L \xrightarrow{a'} DT)$
    - $I'^i_{k-1} = I^i_{k-1} \cup (L \xrightarrow{a'} DT)$ *and* $\omega^i_{k-1}(L \xrightarrow{a'} DT) = (A \xrightarrow{a} DT)$
- *for each* $(M \xrightarrow{b} DV) \in I^i_k$ *such that* $\omega^i_k(M \xrightarrow{b} DV) = (A \xrightarrow{a} DT)$, $\omega^{k-1}_k(M) = L$ *and* $\omega^{k-1}_k(DV) = DT$
  - $I'^{k-1}_k = I^{k-1}_k \cup (M \xrightarrow{b} DV)$ *and* $\omega^{k-1}_k(M \xrightarrow{b} DV) = (L \xrightarrow{a'} DT)$

Spec.	*Input*	*Output*
$\mathfrak{S}_i$	$A \xrightarrow{a^{\triangle p}} DT$	$A \xrightarrow{a^{\triangle p}} DT$
$\vdots$	ω^i_{k-1}, ω^i_k	ω^i_{k-1}, ω^i_k
$\mathfrak{S}_{k-1}$	L DT	$L \xrightarrow{a'^{\triangle 1}} DT$
$\mathfrak{S}_k$	$M \xrightarrow{b^{\triangle 0}} DV$	$M \xrightarrow{b^{\triangle 0}} DV$

The replication rule rac_2 adds to metalevel $k-1$ a replica with single-potency 1 of an atomic constraint with single-potency p at metalevel i, as follows:

Definition 45 (Replication rule rac_2 for atomic constraints)**.** *Given a deep metamodelling stack with length l, for all $1 \leq i < k \leq l$ and $k \geq i+2$:*

- *for each $(A \xrightarrow{a} N) \in T^k_i$ and $(\pi, \delta) \in C^k_i$ where $\delta(\alpha^\Omega(\pi)) = (A \xrightarrow{a} N)$*
  - *for each $(L \xrightarrow{b} Y) \in (T^k_{k-1})$ such that $\omega^i_{k-1}(L \xrightarrow{b} Y) = (A \xrightarrow{a} N)$*
    - $C'^k_{k-1} = C^k_{k-1} \cup (\pi, \delta')$ *where* $\delta'(\alpha^\Omega(\pi)) = (L \xrightarrow{b} Y)$

Spec.	*Input*	*Output*
$\mathfrak{S}_i$	$A \xrightarrow[a]{\pi^{\triangle p}} N$	$A \xrightarrow[a]{\pi^{\triangle p}} N$
$\vdots$	ω^i_{k-1}, ω^i_k	ω^i_{k-1}, ω^i_k
$\mathfrak{S}_{k-1}$	$L \xrightarrow[b]{} Y$	$L \xrightarrow[b]{\pi^{\triangle 1}} Y$
$\mathfrak{S}_k$	$M \xrightarrow{c} Z$	$M \xrightarrow{c} Z$

Note that the rule rac_2 for the replication of atomic constraints is proposed as a proof-of-concept only. This is because this rule is designed to work with the predicates having arities $1 \circlearrowleft^{a}$ and $1 \xrightarrow{a} 2$, e.g., `[irreflexive]` and `[mult(m,n)]` from the signature Ω (see Table 4.2). However, predicates may have arbitrary arities and semantics which may not enable replication of atomic constraints at all. The conditions under which a predicate enables replication of atomic constraints is outside the scope of this work and will be investigated in future work (see Section 4.7).

According to this layering, the application of the rules adds a replica of a reference only *after* it adds a replica of a clabject. This ensures that the rule which adds a replica of a reference matches both clabjects with multi-potency and their instances as well as clabjects with single-potency and their replicas. Moreover, this ensures that the replica of the reference has as source and target an instance of the considered clabject with multi-potency or a replica of the considered clabject with single-potency. The layering of rules for attributes and atomic constraints follow the same rationale.

The extraction rule e_3 projects out the types at each metalevel i and the corresponding instances at metalevel $i+1$ as the elements in each specification of the target partial double metamodelling stack, as follows:

Definition 46 (Extraction rule e_3)**.** *Given a deep metamodelling stack with length l, a double metamodelling stack with length l is extracted as follows:*

- $\mathfrak{S}_1 = (T_1^2, C_1^2, \lambda_1)$
- *for all* $2 \leq i \leq l-1$, $\mathfrak{S}_i = (T_i^{i+1} \cup I_i^{i-1}, C_i^{i+1}, \lambda_i, \omega_i^{i-1})$
- $\mathfrak{S}_l = (I_l^{l-1}, \lambda_l, \omega_l^{l-1})$

Example 32 (Flattening of a deep metamodelling stack)**.** *Building upon Example 31, Figures 4.20(b), (c) and (d) show the specifications $\mathfrak{S}_1$, $\mathfrak{S}_2$ and $\mathfrak{S}_3$ of the deep metamodelling stack, after the application of the replication rules. Moreover, Figure 4.20(c) shows the replicated elements in green colour. Note that the attribute* scroll*, the data type* Boolean *and the corresponding instances are omitted from Figure 4.20 due to space constraints.*

Firstly, the application of ra_1 adds to $\mathfrak{T}_2^3$ the attributes nameMap *and* nameTable *with single-potency $\triangle 1$. Moreover, it adds the following mappings to the ontological typing morphism ω_3^2:*

$\omega_3^2(\textit{nameMapUAM}) = \textit{nameMap}$
$\omega_3^2(\textit{nameTableUAM}) = \textit{nameTable}$
$\omega_3^2(\textit{“UAMCampus”}) = \omega_3^2(\textit{“UAMProfs”}) = \textit{String}$

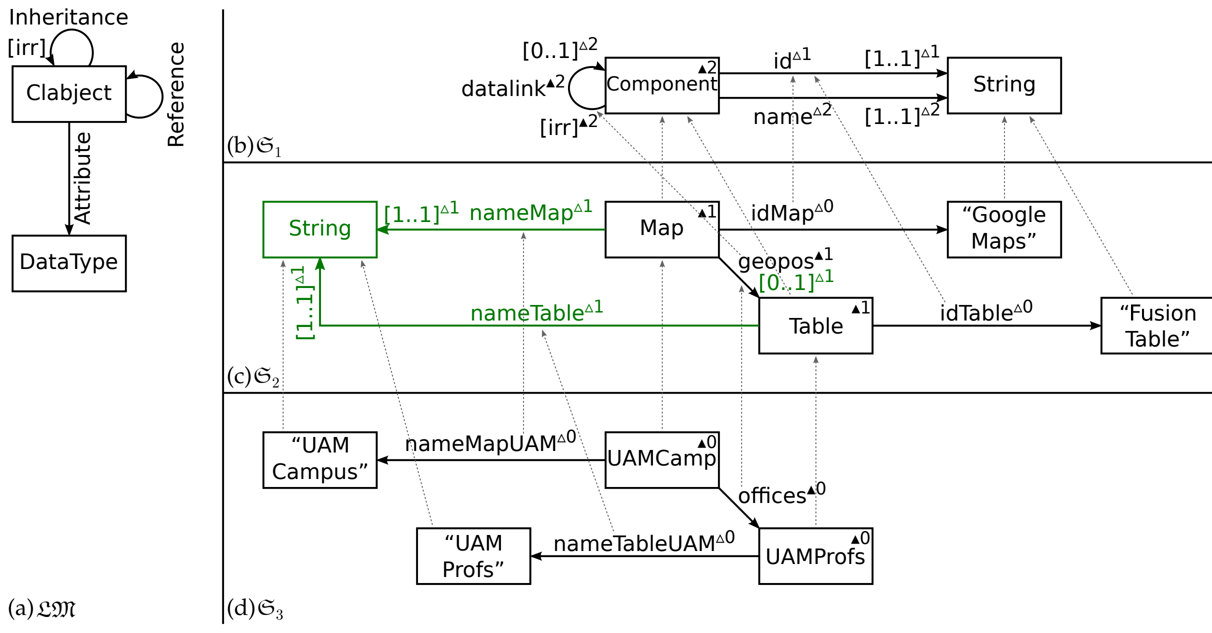


Figure 4.20: The specifications $\mathfrak{LM}$, $\mathfrak{S}_1$, $\mathfrak{S}_2$ and $\mathfrak{S}_3$ together with the ontological typing morphisms ω_2^1 and ω_3^2, after the application of the replication rules

Secondly, the application of rac_2 *adds to* $\mathfrak{T}_2^3$ *the atomic constraints* (`[mult(0,1)]`, δ_1), (`[mult(1,1)]`, δ_2) *and* (`[mult(1,1)]`, δ_3) *with single-potency* $\triangle 1$ *on the reference* **geopos** *and the attributes* **nameMap** *and* **nameTable**, *respectively.*

Figures 4.22(b), (c) and (d) show the specifications $\mathfrak{S}_1$, $\mathfrak{S}_2$ *and* $\mathfrak{S}_3$ *of the partial double metamodelling stack resulting from the application of the extraction rule. Moreover, Figure 4.21(b) shows the discarded elements in red colour.*

The application of e_3 *discards from* $\mathfrak{S}_1$ *the atomic constraints* (`[mult(0,1)]`, δ_1) *and* (`[mult(1,1)]`, δ_4) *on* **datalink** *and* **name**, *respectively. In this way, these atomic constraints are not evaluated at metalevel* 2. *Moreover, it discards from* $\mathfrak{S}_1$ *the attribute* **name**. *In this way, it is not possible to instantiate* **name** *at metalevel* 2.

The presented flattening of the deep characterisation enables the transformation of a deep metamodelling stack into a partial double metamodelling stack. Obviously, part of the deep characterisation information is lost in the transformation. For instance, in Example 31, the multi-potency $\blacktriangle 2$ on the elements Component and datalink in $\mathfrak{S}_1$ forbids that these elements are ontologically typed by elements in a possible specification $\mathfrak{S}_4$ or below. In Example 32, in contrast, a possible specification $\mathfrak{S}_4$ may include elements which are ontologically typed by elements in $\mathfrak{S}_3$.

In addition to the flattening of the deep characterisation, it is possible to define the flattening of the double linguistic/ontological conformance which enables the transformation of a partial double metamodelling stack into a traditional metamodelling stack. This could be done by adding the specification $\mathfrak{LM}$ on top of the ontological stack, and adding a replica of all elements in $\mathfrak{LM}$ in all the specifications $\mathfrak{S}_i$, for all $i \leq l - 2$.

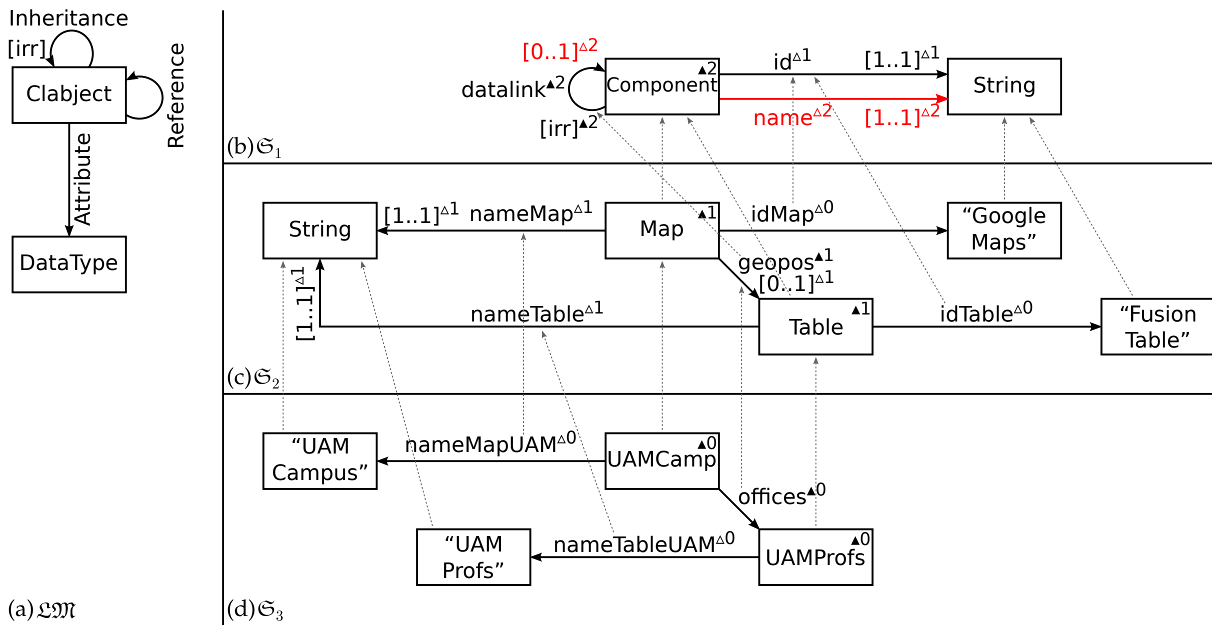


Figure 4.21: The specifications $\mathfrak{LM}$, $\mathfrak{S}_1$, $\mathfrak{S}_2$ and $\mathfrak{S}_3$ together with the ontological typing morphisms ω_2^1 and ω_3^2, before the application of the extraction rule

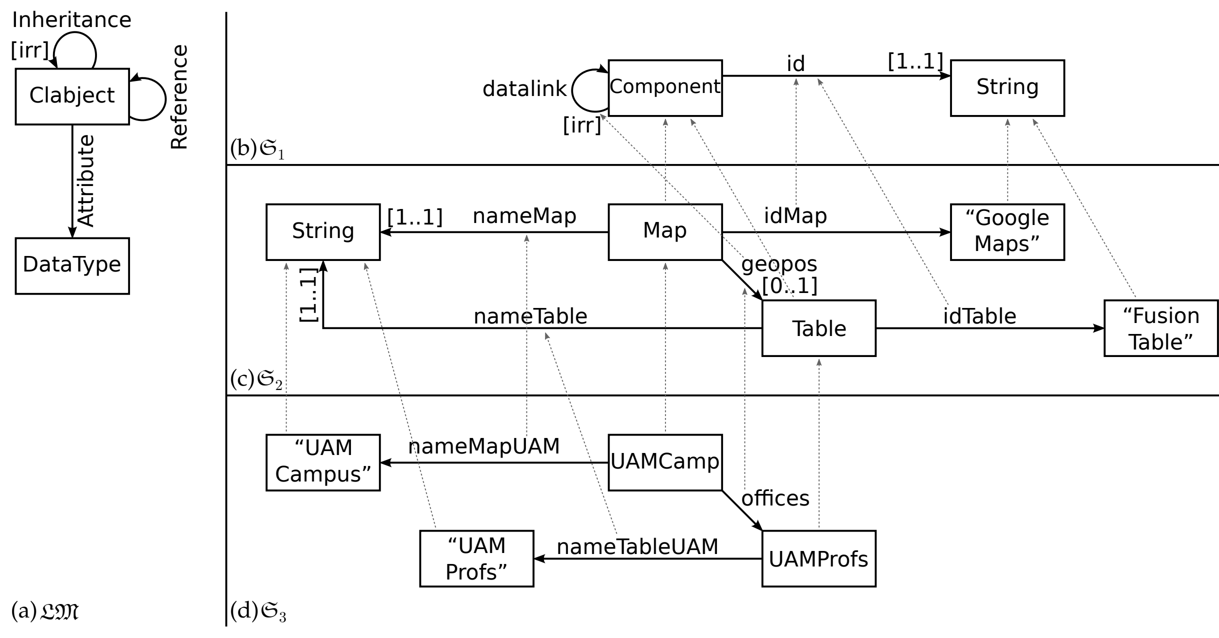


Figure 4.22: The specifications $\mathfrak{LM}$, $\mathfrak{S}_1$, $\mathfrak{S}_2$ and $\mathfrak{S}_3$ together with the ontological typing morphisms ω_2 and ω_3, after the application of the extraction rule

4.6 Related work

Multi-level metamodelling

Deep metamodelling is a relatively new technique, and some of its aspects are still debated in the literature. A first strand of research focuses on multi-level metamodelling.

Early forms of multi-level metamodelling can be traced back to knowledge-based systems like Telos [65] and deductive object base managers like ConceptBase [52].

More recent forms include the works in [6, 23, 41]. In [41], MOF is extended with multiple metalevels to enable XML-based code generation. Nivel [6] is a double metamodelling framework based on the weighted constraint rule language (WCRL). XMF [23] is a language-driven development framework allowing an arbitrary number of metalevels.

Another form of multi-level metamodelling can be achieved through powertypes [42, 72], since instances of powertypes are also subtypes of another type and hence retain both a type and an instance facet. Multi-level metamodelling can also be emulated through stereotypes [71], although this is not a general modelling technique since it relies on UML to emulate the extension of its metamodel. The interested reader can consult [10] for a thorough comparison of potencies, powertypes and stereotypes.

In contrast to our approach, none of the above mentioned works support deep characterisation; i.e., the ability to describe structure and express constraints for metalevels below the adjacent one.

Deep characterisation

A second strand of research focuses on deep characterisation. Deep characterisation through potency is included in the works in [5, 11, 27, 46, 57]. DeepJava [57] is a superset of Java which extends the object-oriented programming paradigm to feature an unbounded number of metalevels. The work in [46] describes the problems arising from the way in which connectors (e.g., associations, links, generalisations, etc.) are supported in mainstream modelling languages such as UML and why they are not suitable for deep metamodelling. The work in [11] presents a prototype implementation of a modelling infrastructure which provides built-in support for multiple ontological as well as linguistic metalevels. The work in [5] proposes a deep metamodelling framework which extends the basic notion of clabject for handling connector inheritance and instantiation. metaDepth [27] is a deep metamodelling framework which supports potency, double linguistic/ontological typing and linguistic extension.

While these works agree on that clabjects are instantiated using the multi-potency semantics, they differ in other design decisions. Firstly, some works are ambiguous about the instantiation semantics for associations. In [57], the associations can be represented as Java references; hence we interpret that they are instantiated using the single-potency semantics. In [46], the connectors are explicitly represented as clabjects but their instantiation semantics is not discussed; hence we interpret that they are

instantiated using the multi-potency semantics. Secondly, not all works adhere to *strict* ontological metamodelling. In [5], the ontological type of an association does not need to be in the adjacent metalevel above, but several metalevels above. Note that our single-potency semantics makes it possible to retain strict metamodelling for associations through a flattening construction that replicates these associations. Finally, some works differ in how they tackle potency on constraints and methods. Potency on constraints is not explicitly shown in [11] and not considered in [5], whereas potency on methods is only supported by DeepJava and METADEPTH.

Table 4.6 shows a summary of the particular semantics for deep characterisation implemented by the above mentioned works and compares it with the semantics supported by our formalisation. It is worth noting that no tool recognises the fact that multiplicity constraints are constraints as well and hence can have a potency.

Table 4.6: Comparison of different deep characterisation semantics

	Clabjects	**Associations**	**Strictness**	**Constraints**	**Mult. constraints**
DeepJava [57]	▲	△	yes	△	N.A.
Atkinson et al. [11]	▲	▲	yes	▲	▲1
Aschauer et al. [5]	▲	▲	no	N.A.	▲1
METADEPTH [27]	△, ▲	△, ▲	yes	△	▲1
DPF formalisation	△, ▲	△, ▲	yes	△, ▲	△, ▲

4.7 Conclusion and future work

In this chapter, we presented a formal approach to deep metamodelling based on DPF. Firstly, we illustrated the limitations of traditional metamodelling through an example in the domain of component-based web applications. Secondly, we introduced deep metamodelling through the same example. Thirdly, we defined double linguistic/ontological typing and linguistic extension in view of DPF. Fourthly, we formalised deep characterisation and defined two different semantics for potency, namely multi- and single-potency. Finally, we showed how to flatten deep characterisation by transforming a deep metamodelling stack into a double metamodelling stack.

To the best of our knowledge, this work is the first attempt to clarify and formalise some aspects of the semantics of deep metamodelling. In particular, this work explains different semantic variation points available for deep metamodelling, points out new possible semantics, currently unexplored in practice, as well as classifies the existing tools according to these options. The findings of this work have already been submitted to a journal for evaluation.

In future work, we will investigate the effects of overriding the potency of a clabject using inheritance, as this may lead to additional contradictory combinations of potencies.

CHAPTER 5

Conclusion

This thesis provides a formalisation of concepts in MDE based on DPF, a formal diagrammatic specification framework which was already under development before this work was initiated. In particular, this thesis aims to consolidate DPF and provide a formalisation of two novel techniques in MDE, namely model versioning and deep metamodelling.

In Chapter 1, we introduced MDE along with a discussion regarding some of its fundamental concepts, techniques and standards.

In Chapter 2, we outlined DPF along with a formalisation of some of the fundamental concepts in MDE. DPF is an adaptation of the categorical sketch formalism, where the constraining constructs of modelling languages are represented by user-defined signatures in a more intuitive and adequate way. In particular, DPF is an extension of the Generalised Sketches formalism and aims to combine mathematical rigour with diagrammatic modelling.

Chapter 2 is an adaptation of the formalisation of modelling and model transformation published in [79, 80, 82, 84]. Compared to the previous work, the specification transformation is extended to support deleting transformation rules. Moreover, the embedding of specification entailments is also revised to adopt deleting transformation rules.

In Chapter 3, we described a formal approach to model versioning based on DPF. Firstly, we defined the identification of commonalities and calculation of differences as pullback and pushout constructions, respectively. Secondly, we defined the representation of differences; i.e., the information added, deleted and renamed, as a set of annotations which are specified by means of a tag signature. Thirdly, we introduced a synchronisation procedure which includes normalisation, conflict detection

and conflict resolution. Specification entailments are adopted to describe properties of the semantic interpretation of predicates of a signature. The normalisation of a specification is then formalised as the embedding of these specification entailments to obtain the normal form of a specification. Moreover, transformation rules are used to represent conflicts and, when applicable, their resolution patterns. The conflict detection and resolution are then formalised as the application of these transformation rules. Note that the approach handles atomic constraints in all the steps of the synchronisation, including normalisation, conflict detection and conflict resolution.

To the best of our knowledge, this work is the first attempt to clarify each step of a work cycle in a centralised approach to optimistic model versioning; i.e., checkout a local copy, make modifications on a local copy, synchronise a local copy, resolve conflicts and commit modifications to a repository. Moreover, this work also constitutes the first attempt to formalise and illustrate constraint-awareness in model versioning.

Chapter 3 further develops the formal approach to model versioning published in [78, 81]. Compared to the previous work, the theoretical foundation and the underlying techniques are extended to handle constraints. Moreover, new examples are added to illustrate how model merging, conflict detection and conflict resolution handle constraints. The findings of this work have already been submitted to a journal for evaluation.

In Chapter 4, we presented a formal approach to deep metamodelling based on DPF. Firstly, we illustrated the limitations of traditional metamodelling through an example in the domain of component-based web applications. Secondly, we introduced deep metamodelling through the same example. Thirdly, we defined double linguistic/ontological typing and linguistic extension in view of DPF. Fourthly, we formalised deep characterisation and defined two different semantics for potency, namely multi- and single-potency. Finally, we showed how to flatten deep characterisation by transforming a deep metamodelling stack into a double metamodelling stack.

To the best of our knowledge, this work is the first attempt to clarify and formalise some aspects of the semantics of deep metamodelling. In particular, this work explains different semantic variation points available for deep metamodelling, points out new possible semantics, currently unexplored in practice, as well as classifies the existing tools according to these options. The findings of this work have already been submitted to a journal for evaluation.

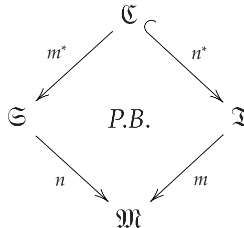
In this thesis, the formalisation of model versioning and deep metamodelling have been treated as two independent research strands. Considering that the formal approaches to model versioning and deep metamodelling share DPF as the formal underpinning, model versioning in the context of deep metamodelling may represent a natural next step in this research.

DPF is a general and open framework still under development and with potential applications in many areas of software engineering and informatics. In this thesis, DPF has been adopted as a formal foundation for two novel techniques in MDE. The intention of writing a monograph was to consolidate and present the current state of the development of DPF, especially by adopting precise and consistent terminology and notation as well as practical examples in MDE. We hope that we have convinced the reader that DPF has the potential to support the foundation and the further development of MDE.

APPENDIX A

This appendix details the constructions adopted in this thesis.

Proposition 5 (Pullback)**.** *Given specifications* $\mathfrak{M} = (M, C^{\mathfrak{M}} : \Sigma)$, $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$, $\mathfrak{T} = (T, C^{\mathfrak{T}} : \Sigma)$ *and injective specification morphisms* $n : \mathfrak{S} \to \mathfrak{M}$, $m : \mathfrak{T} \to \mathfrak{M}$, *one can construct a specification* $\mathfrak{C} = (C, C^{\mathfrak{C}} : \Sigma)$ *and an injective specification morphism* $m^* : \mathfrak{C} \to \mathfrak{S}$ *such that* $\mathfrak{C} \sqsubseteq \mathfrak{T}$ *and the resulting diagram is commutative and a pullback in the category* $\mathbf{Spec}(\Sigma)$*.*



The graph C is defined as follows:

$$C_N := \{X \in T_N \mid m_N(X) \in n_N(S_N)\}$$
$$C_A := \{f \in T_A \mid m_A(f) \in n_A(S_A)\}$$
$$src^C(f) := src^T(f) \text{ for all } f \in C_A$$
$$trg^C(f) := trg^T(f) \text{ for all } f \in C_A$$

Moreover, the graph homomorphism $m^* : C \to S$ *is defined as follows:*

$$m^*_N(X) := n_N^{-1}(m_N(X)) \text{ for all } X \in C_N$$
$$m^*_A(f) := n_A^{-1}(m_A(f)) \text{ for all } f \in C_A$$

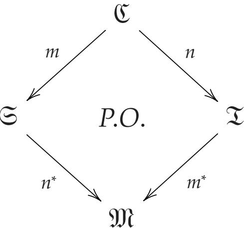
Finally, the set of atomic constraints $C^{\mathfrak{C}}$ is defined as follows:

$$C^{\mathfrak{C}} := \{(\pi, \delta) \in C^{\mathfrak{T}} \mid \exists(\pi, \epsilon) \in C^{\mathfrak{S}} \text{ with } \delta; m = \epsilon; n\}$$

Remark 11 (Uniqueness of pullback)**.** *The pullback $(\mathfrak{C}, m^* : \mathfrak{C} \to \mathfrak{S}, n^* : \mathfrak{C} \hookrightarrow \mathfrak{T})$ in Proposition 5 is unique since the specification morphism n^* is an inclusion.*

In the following, the notation $S.\mathsf{x}$ refers to the element x in the specification $\mathfrak{S}$, where $\mathfrak{S}$ is considered the name (unique identifier) of the specification. This notation is used to resolve possible name conflicts; i.e., to ensure disjoint union.

Proposition 6 (Pushout)**.** *Given specifications $\mathfrak{C} = (C, C^{\mathfrak{C}} : \Sigma)$, $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$, $\mathfrak{T} = (T, C^{\mathfrak{T}} : \Sigma)$ and injective specification morphisms $m : \mathfrak{C} \to \mathfrak{S}$, $n : \mathfrak{C} \to \mathfrak{T}$, one can construct a specification $\mathfrak{M} = (M, C^{\mathfrak{M}} : \Sigma)$ and injective specification morphisms $n^* : \mathfrak{S} \to \mathfrak{M}$, $m^* : \mathfrak{T} \to \mathfrak{M}$ such that the resulting diagram is commutative and a pushout in the category* **Spec**(Σ).



The graph M is defined as follows:

$$M_N := \{S.X \mid X \in S_N \setminus m_N(C_N)\} \cup C_N \cup \{T.X \mid X \in T_N \setminus n_N(C_N)\}$$
$$M_A := \{S.g \mid g \in S_A \setminus m_A(C_A)\} \cup C_A \cup \{T.g \mid g \in T_A \setminus n_A(C_A)\}$$
$$src^M(f) := \begin{cases} Y, & \text{if } f = S.g, m_N(Y) = src^S(g) \in m_N(C_N) \\ S.X, & \text{if } f = S.g, X = src^S(g) \notin m_N(C_N) \\ src^C(f), & \text{if } f \in C_A \\ Y, & \text{if } f = T.g, n_N(Y) = src^T(g) \in n_N(C_N) \\ T.X, & \text{if } f = T.g, X = src^T(g) \notin n_N(C_N) \end{cases}$$

$trg^M(f)$ *is defined analogously*

Moreover, the graph homomorphisms $n^* : S \to M$, $m^* : T \to M$ are defined as follows:

$$n^*_N(X) := \begin{cases} Y, & \text{if } X = m_N(Y) \in m_N(C_N) \\ S.X, & \text{if } X \notin m_N(C_N) \end{cases}$$

$$m^*_N(X) := \begin{cases} Y, & \text{if } X = n_N(Y) \in n_N(C_N) \\ T.X, & \text{if } X \notin n_N(C_N) \end{cases}$$

$$n^*_A(f) := \begin{cases} g, & \text{if } f = m_A(g) \in m_A(C_A) \\ S.f, & \text{if } f \notin m_A(C_A) \end{cases}$$

$$m^*_A(f) := \begin{cases} g, & \text{if } f = n_A(g) \in n_A(C_A) \\ T.f, & \text{if } f \notin n_A(C_A) \end{cases}$$

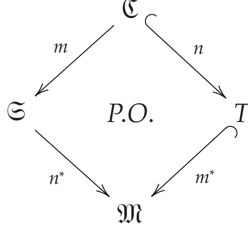
Finally, the set of atomic constraints $C^{\mathfrak{M}}$ is defined as follows:

$$C^{\mathfrak{M}} := \{(\pi, \delta; n^*) \mid (\pi, \delta) \in C^{\mathfrak{S}}\} \cup \{(\rho, \epsilon; m^*) \mid (\rho, \epsilon) \in C^{\mathfrak{T}}\}$$

Remark 12 (Uniqueness of representatives)**.** *The Y's and g's in Proposition 6 are uniquely determined since the specification morphisms m and n are assumed to be injective.*

In case of inclusion specification morphisms, the pushout construction can be simplified:

Proposition 7 (Pushout for inclusion specification morphisms)**.** *If $\mathfrak{C} \sqsubseteq \mathfrak{T}$ there will not be name conflicts between $\mathfrak{C}$ and $\mathfrak{T}$. In this case, the specification morphism m^* is assumed to be inclusion, which simplifies the construction of $\mathfrak{M}$.*



The graph M is defined as follows:

$$M_N := \{S.X \mid X \in S_N \setminus m_N(C_N)\} \cup T_N$$
$$M_A := \{S.g \mid g \in S_A \setminus m_A(C_A)\} \cup T_A$$
$$src^M(f) := \begin{cases} Y, & \text{if } f = S.g, m_N(Y) = src^S(g) \in m_N(C_N) \\ S.X, & \text{if } f = S.g, X = src^S(g) \notin m_N(C_N) \\ src^T(f), & \text{if } f \in T_A \end{cases}$$

$trg^M(f)$ is defined analogously

Finally, the set of atomic constraints $C^{\mathfrak{M}}$ is defined as follows:

$$C^{\mathfrak{M}} := \{(\pi, \delta; n^*) \mid (\pi, \delta) \in C^{\mathfrak{S}}\} \cup C^{\mathfrak{T}}$$

Proposition 8 (Pushout for annotated specifications)**.** *Given specifications $\mathfrak{C} = (C, C^{\mathfrak{C}} : \Sigma, A^{\mathfrak{C}} : \Delta)$, $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma, A^{\mathfrak{S}} : \Delta)$, $\mathfrak{T} = (T, C^{\mathfrak{T}} : \Sigma, A^{\mathfrak{T}} : \Delta)$ and injective specification morphisms $m : \mathfrak{C} \to \mathfrak{S}$, $n : \mathfrak{C} \to \mathfrak{T}$, one can construct a specification $\mathfrak{M} = (M, C^{\mathfrak{M}} : \Sigma, A^{\mathfrak{M}} : \Delta)$ and injective specification morphisms $n^* : \mathfrak{S} \to \mathfrak{M}$, $m^* : \mathfrak{T} \to \mathfrak{M}$ such that the resulting diagram is commutative and a pushout in the category* **Spec**$(\Sigma \cup \Delta)$.

The graph M, the graph homomorphisms $n^* : S \to M$, $m^* : T \to M$, and the set of atomic constraints $C^{\mathfrak{M}}$ are defined as is Proposition 6.

Finally, the set of atomic constraints $A^{\mathfrak{M}}$ is defined as follows:

$$A^{\mathfrak{M}} := \{(\theta, \delta; n^*) \mid (\theta, \delta) \in A^{\mathfrak{S}}\} \cup \{(\upsilon, \epsilon; m^*) \mid (\upsilon, \epsilon) \in A^{\mathfrak{T}}\}$$

Remark 13 (Identification of atomic constraints and annotations)**.** *Two atomic constraints $(\pi, \delta) \in C^{\mathfrak{S}}$ and $(\pi, \epsilon) \in C^{\mathfrak{T}}$ such that $\delta; m^* = \epsilon; n^*$ are identified and will be mapped to the same atomic constraint $(\pi, \delta; m^*) = (\pi, \epsilon; n^*) \in C^{\mathfrak{M}}$; i.e., for all atomic constraints $(\rho, \gamma) \in C^{\mathfrak{C}}$ we obtain just one atomic constraint $(\rho, \gamma; n; m^*) = (\rho, \gamma; m; n^*) \in C^{\mathfrak{M}}$. The same applies to annotations $(\theta, \delta) \in A^{\mathfrak{S}}$.*

Definition 47 (Partial map)**.** *A partial map $f : A \multimap B$ between two sets A and B is given by the domain of definition $dom(f) \subseteq A$ and a total map $f : dom(f) \to B$. For any subset $A_0 \subseteq A$, the image of the subset A_0 under f is defined as $f(A_0) = \{f(a) \mid a \in A_0 \text{ and } a \in dom(f)\} \subseteq f(A) \subseteq B$. For any subset $B_0 \subseteq B$, the inverse image of the subset B_0 under f is defined as $f^{-1}(B_0) = \{a \in dom(f) \mid f(a) \in B_0\} \subseteq f^{-1}(B) \subseteq A$. The composition of two partial maps $f : A \multimap B$ and $g : B \multimap C$ is defined by $dom(f; g) = f^{-1}(dom(g)) \subseteq dom(f)$ and $(f; g)(a) = g(f(a))$, for all $a \in dom(f; g)$.*

It is straightforward to check that: the composition of partial maps is associative; for any subset $C_0 \subseteq C$ we have $(f; g)^{-1}(C_0) = g^{-1}(f^{-1}(C_0))$; for any subset $B_0 \subseteq B$ we have $f(f^{-1}(B_0)) \subseteq B_0$ and hence $f(dom(f; g)) \subseteq dom(g)$.

Definition 48 (Partial order over partial maps)**.** *A partial order $\sqsubseteq$ over the set of all partial maps from the set A to the set B can be defined as: given two partial maps $f, g : A \multimap B$, $f \sqsubseteq g$ if and only if $dom(f) \subseteq dom(g)$ and $f(a) = g(a)$, for all $a \in dom(f)$.*

Bibliography

[1] Marcus Alanen and Ivan Porres. Difference and Union of Models. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *Proceedings of UML 2003: 6th International Conference on The Unified Modeling Language, Modeling Languages and Applications*, volume 2863 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2003. ISBN 978-3-540-20243-1. DOI 10.1007/978-3-540-45221-8_2.

[2] Alloy. *Project Web Site*. `http://alloy.mit.edu/community/`.

[3] Michał Antkiewicz and Krzysztof Czarnecki. Design Space of Heterogeneous Synchronization. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Proceedings of GTTSE 2007: Generative and Transformational Techniques in Software Engineering II, International Summer School*, volume 5235 of *Lecture Notes in Computer Science*, pages 3–46. Springer, 2008. ISBN 978-3-540-88642-6. DOI 10.1007/978-3-540-88643-3_1.

[4] Apache Subversion. *Project Web Site*. `http://subversion.apache.org/`.

[5] Thomas Aschauer, Gerd Dauenhauer, and Wolfgang Pree. Multi-level Modeling for Industrial Automation Systems. In *Proceedings of EUROMICRO 2009: 35th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 490–496. IEEE Computer Society, 2009. ISBN 978-0-7695-3784-9. DOI 10.1109/SEAA.2009.46.

[6] Timo Asikainen and Tomi Männistö. Nivel: a metamodelling language with a formal semantics. *Software and Systems Modeling*, 8(4): 521–549, 2009. DOI 10.1007/s10270-008-0103-2.

[7] Colin Atkinson and Thomas Kühne. Rearchitecting the UML infrastructure. *ACM Transactions on Modeling and Computer Simulation*, 12 (4):290–321, 2002. DOI 10.1145/643120.643123.

[8] Colin Atkinson and Thomas Kühne. Profiles in a strict metamodeling framework. *Science of Computer Programming*, 44(1):5–22, 2002. DOI 10.1016/S0167-6423(02)00029-1.

[9] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003. DOI 10.1109/MS.2003.1231149.

[10] Colin Atkinson and Thomas Kühne. Reducing accidental complexity in domain models. *Software and Systems Modeling*, 7(3):345–359, 2008. DOI 10.1007/s10270-007-0061-0.

[11] Colin Atkinson, Matthias Gutheil, and Bastian Kennel. A Flexible Infrastructure for Multilevel Language Engineering. *IEEE Transactions on Software Engineering*, 35(6):742–755, 2009. DOI 10.1109/TSE.2009.31.

[12] Michael Barr and Charles Wells. *Category Theory for Computing Science (2nd Edition)*. Prentice Hall, 1995. ISBN 978-0-13-323809-9.

[13] Øyvind Bech. DPF Editor: A Multi-Layer Modelling Environment for Diagram Predicate Framework in Eclipse. Master's thesis, Department of Informatics, University of Bergen, Norway, May 2011.

[14] Øyvind Bech and Dag Viggo Lokøen. *DPF to SHIP Validator Proof-of-Concept Transformation Engine*. `http://dpf.hib.no/code/transformation/dpf_to_shipvalidator.py`.

[15] Bergen University College and University of Bergen. *Diagram Predicate Framework Web Site*. `http://dpf.hib.no/`.

[16] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171–188, 2005. DOI 10.1007/s10270-005-0079-0.

[17] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proceedings of ASE 2001: 16th IEEE International Conference on Automated Software Engineering*, pages 273–280, 2001. ISBN 978-0-7695-1426-0. DOI 10.1109/ASE.2001.989813.

[18] Artur Boronat and José Meseguer. Algebraic Semantics of OCL-Constrained Metamodel Specifications. In Manuel Oriol, Bertrand Meyer, Wil Aalst, John Mylopoulos, Michael Rosemann, Michael J. Shaw, and Clemens Szyperski, editors, *Proceedings of TOOLS 2009: 47th International Conference on Objects, Components, Models and Patterns*, volume 33 of *Lecture Notes in Business Information Processing*, pages 96–115. Springer, 2009. ISBN 978-3-642-02571-6. DOI 10.1007/978-3-642-02571-6_7.

[19] Petra Brosch, Gerti Kappel, Martina Seidl, Konrad Wieland, Manuel Wimmer, Horst Kargl, and Philip Langer. Adaptable Model Versioning in Action. In Gregor Engels, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2010*, volume 161 of *Lecture Notes in Informatics*, pages 221–236. GI, 2010. ISBN 978-3-88579-255-0.

[20] Cambridge. *Dictionaries Online*. http://dictionary.cambridge.org.

[21] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, October 2007. DOI 10.5381/jot.2007.6.9.a9.

[22] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. Managing Model Conflicts in Distributed Development. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Proceedings of MoDELS 2008: 11th International Conference on Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2008. ISBN 978-3-540-87874-2. DOI 10.1007/978-3-540-87875-9_23.

[23] Tony Clark, Paul Sammut, and James Willans. *Applied Metamodelling: A Foundation for Language Driven Development (2nd Edition)*. Ceteva, 2008.

[24] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*, 2007. Springer. ISBN 978-3-540-71940-3. DOI 10.1007/978-3-540-71999-1.

[25] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, June 2000. ISBN 978-0-201-30977-5.

[26] Marcos Aurélio Almeida da Silva, Alix Mougenot, Xavier Blanc, and Reda Bendraou. Towards Automated Inconsistency Handling in Design Models. In Barbara Pernici, editor, *Proceedings of CAiSE 2010: 22nd International Conference on Advanced Information Systems Engineering*, volume 6051 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 2010. ISBN 978-3-642-13093-9. DOI 10.1007/978-3-642-13094-6_28.

[27] Juan de Lara and Esther Guerra. Deep Meta-modelling with MetaDepth. In Jan Vitek, editor, *Proceedings of TOOLS 2010: 48th International Conference on Objects, Components, Models and Patterns*, volume 6141 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2010. ISBN 978-3-642-13952-9. DOI 10.1007/978-3-642-13953-6_1.

[28] Zinovy Diskin. *Practical foundations of business system specifications*, chapter Mathematics of UML: Making the Odysseys of UML less

dramatic, pages 145–178. Kluwer Academic Publishers, 2003. ISBN 978-1-4020-1480-2.

[29] Zinovy Diskin and Jürgen Dingel. Mappings, Maps and Tables: Towards Formal Semantics for Associations in UML2. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Proceedings of MoDELS 2006: 9th International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 230–244. Springer, 2006. ISBN 978-3-540-45772-5. DOI 10.1007/11880240_17.

[30] Zinovy Diskin and Boris Kadish. Generic Model Management. In *Encyclopedia of Database Technologies and Applications*, pages 258–265. Idea Group, 2005. ISBN 978-1-59140-560-3.

[31] Zinovy Diskin and Uwe Wolter. A Diagrammatic Logic for Object-Oriented Visual Modeling. In *Proceedings of ACCAT 2007: 2nd Workshop on Applied and Computational Category Theory*, volume 203/6 of *Electronic Notes in Theoretical Computer Science*, pages 19–41. Elsevier, 2008. DOI 10.1016/j.entcs.2008.10.041.

[32] Eclipse Modeling Framework. *Project Web Site*. `http://www.eclipse.org/emf/`.

[33] Eclipse Platform. *Project Web Site*. `http://www.eclipse.org`.

[34] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools*. World Scientific Publishing Company, 1999. ISBN 978-981-02-4020-2.

[35] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental Theory for Typed Attributed Graph Transformation. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Proceedings of ICGT 2004: 2nd International Conference on Graph Transformations*, volume 3256 of *Lecture Notes in Computer Science*, pages 161–177. Springer, 2004. ISBN 978-3-540-23207-0. DOI 10.1007/978-3-540-30203-2_13.

[36] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, March 2006. ISBN 978-3-540-31187-4. DOI 10.1007/3-540-31188-2.

[37] EMF Compare. *Project Web Site*. `http://www.eclipse.org/emft/projects/compare/`.

[38] José Luiz Fiadeiro. *Categories for Software Engineering*. Springer, May 2004. ISBN 978-3-540-20909-6.

[39] David S. Frankel and John Parodi. *The MDA Journal: Model Driven Architecture Straight From The Masters*. Meghan-Kiffer Press, 2004. ISBN 978-0-929652-25-2.

[40] Git. *Project Web Site*. http://git-scm.com/.

[41] Ralf Gitzel, Ingo Ott, and Martin Schader. Ontological Extension to the MOF Metamodel as a Basis for Code Generation. *Computer Journal*, 50(1):93–115, 2007. DOI 10.1093/comjnl/bxl052.

[42] Cesar Gonzalez-Perez and Brian Henderson-Sellers. A powertype-based metamodelling framework. *Software and Systems Modeling*, 5 (1):72–90, 2006. DOI 10.1007/s10270-005-0099-9.

[43] Cesar Gonzalez-Perez and Brian Henderson-Sellers. *Metamodelling for Software Engineering*. Wiley, 2008. ISBN 978-0-470-03036-3.

[44] Graphical Editing Framework. *Project Web Site*. http://www.eclipse.org/gef/.

[45] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004. ISBN 978-0-471-20284-4.

[46] Matthias Gutheil, Bastian Kennel, and Colin Atkinson. A Systematic Approach to Connectors in a Multi-level Modeling Environment. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Proceedings of MoDELS 2008: 11th International Conference on Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 843–857. Springer, 2008. ISBN 978-3-540-87874-2. DOI 10.1007/978-3-540-87875-9_58.

[47] Reiko Heckel. Graph Transformation in a Nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1):187–198, 2006. DOI 10.1016/j.entcs.2005.12.018.

[48] Wolfgang Hesse. More matters on (meta-)modelling: remarks on Thomas Kühne's "matters". *Software and Systems Modeling*, 5(4):387–394, 2006. DOI 10.1007/s10270-006-0033-9.

[49] Dag Hovland, Federico Mancini, and Khalid A. Mughal. The SHIP Validator: An Annotation-Based Content-Validation Framework for Java Applications. Technical Report 389, Department of Informatics, University of Bergen, Norway, September 2009.

[50] James W. Hunt and M. D. McIlroy. An Algorithm for Differential File Comparison. Technical Report 41, Bell Laboratories, Murray Hill, NJ, USA, 1976.

[51] Internet Engineering Task Force. *RFC4122: A Universally Unique IDentifier (UUID) URN Namespace*, July 2005. http://www.ietf.org/rfc/rfc4122.txt.

[52] Matthias Jarke, Rainer Gallersdörfer, Manfred A. Jeusfeld, and Martin Staudt. ConceptBase - A deductive object base for meta data management. *Journal of Intelligent Information Systems*, 4(2):167–192, 1995. DOI 10.1007/BF00961873.

[53] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003. ISBN 978-0-321-19442-8.

[54] Dimitrios S. Kolovos, Louis M. Rose, and Richard Paige. *The Epsilon Book*. http://www.eclipse.org/gmt/epsilon/doc/book/.

[55] Thomas Kühne. Matters of (meta-)modeling. *Software and Systems Modeling*, 5(4):369–385, 2006. DOI 10.1007/s10270-006-0017-9.

[56] Thomas Kühne. Clarifying matters of (meta-)modeling: an author's reply. *Software and Systems Modeling*, 5(4):395–401, 2006. DOI 10.1007/s10270-006-0034-8.

[57] Thomas Kühne and Daniel Schreiber. Can Programming be Liberated from the Two-Level Style? Multi-Level Programming with DeepJava. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of OOPSLA 2007: 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 229–244. ACM, 2007. ISBN 978-1-59593-786-5. DOI 10.1145/1297027.1297044.

[58] Ivan Kurtev, Jean Bézivin, and Frédéric Jouault andPatrick Valduriez. Model-Based DSL Frameworks. In *Proceedings of OOPSLA 2006: 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 602–616. ACM, 2006. ISBN 978-1-59593-491-8. DOI 10.1145/1176617.1176632.

[59] Yuehua Lin, Jeff Gray, and Frédéric Jouault. DSMDiff: A Differentiation Tool for Domain-Specific Models. *European Journal of Information Systems*, 16(4, Special Issue on Model-Driven Systems Development): 349–361, 2007. DOI 10.1057/palgrave.ejis.3000685.

[60] Michael Makkai. Generalized Sketches as a Framework for Completeness Theorems. *Journal of Pure and Applied Algebra*, 115(1):49–79, 179–212, 214–274, 1997. DOI 10.1016/S0022-4049(96)00007-2.

[61] Federico Mancini, Dag Hovland, and Khalid A. Mughal. Investigating the Limitations of Java Annotations for Input Validation. In *Proceedings of ARES 2010: 5th International Conference on Availability, Reliability and Security*. IEEE Computer Society, 2010. ISBN 978-1-4244-5879-0. DOI 10.1109/ARES.2010.29.

[62] Tom Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002. DOI 10.1109/TSE.2002.1000449.

[63] MeTEOriC: Meta-Tool Environments for Model-Oriented Collaborative Web Applications. *Project Web Site*. http://ishtar.ii.uam.es/meteoric/.

[64] Leonardo Murta, Chessman Corrêa, João Gustavo Prudêncio, and Cláudia Werner. Towards odyssey-VCS 2: improvements over a UML-based version control system. In *Proceedings of CVSM 2008: International workshop on Comparison and Versioning of Software Models*, pages 25–30. ACM, 2008. ISBN 978-1-60558-045-6. DOI 10.1145/1370152.1370159.

[65] John Mylopoulos, Alexander Borgida, Matthias Jarke, and Manolis Koubarakis. Telos: Representing Knowledge About Information Systems. *ACM Transactions on Information Systems*, 8(4):325–362, 1990. DOI 10.1145/102675.102676.

[66] Object Management Group. *Web site*. http://www.omg.org.

[67] Object Management Group. *MDA Guide*, June 2003. http://www.omg.org/cgi-bin/doc?omg/03-06-01.

[68] Object Management Group. *Meta-Object Facility Specification*, January 2006. http://www.omg.org/spec/MOF/2.0/.

[69] Object Management Group. *XML Metadata Interchange Specification*, December 2007. http://www.omg.org/spec/XMI/2.1.1/.

[70] Object Management Group. *Object Constraint Language Specification*, February 2010. http://www.omg.org/spec/OCL/2.2/.

[71] Object Management Group. *Unified Modeling Language Specification*, May 2010. http://www.omg.org/spec/UML/2.3/.

[72] James Odell. Power Types. *Journal of Object-Oriented Programming*, 7 (2):8–12, 1994.

[73] Dirk Ohst, Michael Welle, and Udo Kelter. Differences between versions of UML diagrams. In *Proceedings of ESEC/FSE 2003: 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003*, pages 227–236. ACM, 2003. ISBN 978-1-58113-743-5. DOI 10.1145/940071.940102.

[74] OMG Model Driven Architecture. *Web Site*. http://www.omg.org/mda/.

[75] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion (2nd Edition)*. O'Reilly Media, October 2008. ISBN 978-0-596-51033-6.

[76] Iman Poernomo. A Type Theoretic Framework for Formal Metamodelling. In *International Seminar on Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science*, pages 262–298. Springer, 2006. ISBN 978-3-540-35800-8. DOI 10.1007/11786160_15.

[77] José E. Rivera and Antonio Vallecillo. Representing and Operating with Model Differences. In Richard F. Paige, Bertrand Meyer, Wil Aalst, John Mylopoulos, Michael Rosemann, Michael J. Shaw, and Clemens Szyperski, editors, *Proceedings of TOOLS 2008: 46th International Conference on Objects, Components, Models and Patterns*, volume 11 of *Lecture Notes in Business Information Processing*, pages 141–160. Springer, 2008. ISBN 978-3-540-69823-4. DOI 10.1007/978-3-540-69824-1_9.

[78] Alessandro Rossini, Adrian Rutle, Yngve Lamo, and Uwe Wolter. A formalisation of the copy-modify-merge approach to version control in MDE. *Journal of Logic and Algebraic Programming*, 79(7):636–658, 2010. DOI 10.1016/j.jlap.2009.10.003.

[79] Alessandro Rossini, Adrian Rutle, Khalid A. Mughal, Yngve Lamo, and Uwe Wolter. A Formal Approach to Data Validation Constraints in MDE. In Marcel Kyas, Sun Meng, and Volker Stolz, editors, *Proceedings of TTSS 2011: 5th International Workshop on Harnessing Theories for Tool Support in Software*, pages 65–76, September 2011. ISBN 82-7368-371-0.

[80] Adrian Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, Department of Informatics, University of Bergen, Norway, 2010.

[81] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A Category-Theoretical Approach to the Formalisation of Version Control in MDE. In Marsha Chechik and Martin Wirsing, editors,

Proceedings of FASE 2009: 12th International Conference on Fundamental Approaches to Software Engineering, volume 5503 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2009. ISBN 978-3-642-00592-3. DOI 10.1007/978-3-642-00593-0_5.

[82] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A Diagrammatic Formalisation of MOF-Based Modelling Languages. In Manuel Oriol, Bertrand Meyer, Wil Aalst, John Mylopoulos, Michael Rosemann, Michael J. Shaw, and Clemens Szyperski, editors, *Proceedings of TOOLS 2009: 47th International Conference on Objects, Components, Models and Patterns*, volume 33 of *Lecture Notes in Business Information Processing*, pages 37–56. Springer, 2009. ISBN 978-3-642-02571-6. DOI 10.1007/978-3-642-02571-6_4.

[83] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A Formalisation of Constraint-Aware Model Transformations. In David Rosenblum and Gabriele Taentzer, editors, *Proceedings of FASE 2010: 13th International Conference on Fundamental Approaches to Software Engineering*, volume 6013 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2010. ISBN 978-3-642-12028-2. DOI 10.1007/978-3-642-12029-9_2.

[84] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A formal approach to the specification and transformation of constraints in MDE. *Journal of Logic and Algebraic Programming*, To appear.

[85] Ed Seidewitz. What Models Mean. *IEEE Software*, 20(5):26–32, 2003. DOI 10.1109/MS.2003.1231147.

[86] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0 (2nd Edition)*. Addison-Wesley Professional, 2008. ISBN 978-0-321-33188-5.

[87] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. Conflict Detection for Model Versioning Based on Graph Modifications. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *Proceedings of ICGT 2010: 5th International Conference on Graph Transformations*, volume 6372 of *Lecture Notes in Computer Science*, pages 171–186. Springer, 2010. ISBN 978-3-642-15927-5. DOI 10.1007/978-3-642-15928-2_12.

[88] Bernhard Westfechtel. A Formal Approach to Three-Way Merging of EMF Models. In *Proceedings of IWMCP 2010: 1st International Workshop on Model Comparison in Practice*, pages 31–41. ACM, 2010. ISBN 978-1-60558-960-2. DOI 10.1145/1826147.1826155.

[89] Uwe Wolter and Zinovy Diskin. The Next Hundred Diagrammatic Specification Techniques – An Introduction to Generalized Sketches. Technical Report 358, Department of Informatics, University of Bergen, Norway, July 2007.

[90] Uwe Wolter and Zinovy Diskin. From Indexed to Fibred Semantics – The Generalized Sketch File. Technical Report 361, Department of Informatics, University of Bergen, Norway, October 2007.